TURING 图灵程序设计丛书



简约之美
软件设计之道

Simplicity: The Science of Software Development

[美] *Max Kanat-Alexander* 著
余晟 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

优秀的软件设计简单明了。不过很遗憾，如今的计算机程序基本上都很复杂，恐怕无人能够确切知道所有代码都是怎么运转的。这本简明教程旨在帮助读者利用科学规则掌握优秀设计的基础知识，书中给出的法则适用于所有编程语言和软件项目，并且永远有效。

不论是刚入门的程序员、资深软件工程师还是没有技术背景的管理人员，读过本书之后，都将能够理解如何创建靠谱的软件项目计划、确定更好的系统模型和架构。

- 为什么软件设计成了一门缺失的科学
- 软件和优秀软件设计的终极目标
- 确定现在以及将来软件设计的价值所在
- 用真实案例证明系统如何随时间变化而变化
- 好的设计能适应外界尽可能多的变化，而软件自身的变化尽可能少
- 代码越简洁，未来做改动的难度就越低
- 测试越准确，软件性能就越有把握

本书原版提供电子书，有PDF、EPUB、Mobi、APK和DAISY等多种版本供选择，所有版本都是没有数字版权管理的（DRM-free）。读者可登录oreilly.com购买，购书者可享受免费修订版本。



O'REILLY
oreilly.com.cn

封面设计：Karen Montgomery　马冬燕
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604



分类建议　计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn
O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-30238-0
定价：25.00元

TURING 图灵程序设计丛书

简约之美
软件设计之道

Code Simplicity
The Science of Software Development

[美] Max Kanat-Alexander 著

余晟 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北 京

图书在版编目（CIP）数据

简约之美 ：软件设计之道 / （美）卡纳特-亚历山大
(Kanat-Alexander,M.) 著 ；余晟译. -- 北京 ：人民邮
电出版社，2013.1
（图灵程序设计丛书）
书名原文：Code Simplicity : The Science of
Software Development
ISBN 978-7-115-30238-0

Ⅰ. ①简… Ⅱ. ①卡… ②余… Ⅲ. ①软件设计
Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2012)第293703号

内 容 提 要

本书将软件设计作为一门严谨的科学，阐述了开发出优雅简洁的代码所应该遵循的基本原则。作者从为什么以前软件设计没有像数学等学科一样成为一门科学开始入手，道出了软件以及优秀的软件设计的终极目标，并给出了具体的指导规则。

这是一本软件思想著作，适合任何背景、使用任何语言的程序员。

图灵程序设计丛书

简约之美：软件设计之道

◆ 著 [美] Max Kanat-Alexander
译 余 晟
责任编辑 朱 巍

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京天宇星印刷厂印刷

◆ 开本：880×1230 1/32
印张：3.75
字数：94千字 2013年1月第1版
印数：1－4 000册 2013年1月北京第1次印刷

著作权合同登记号 图字：01-2012-7086号

ISBN 978-7-115-30238-0

定价：25.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

目录

第1章 引言



本书大部分篇幅讲解的是“软件设计”，你以前可能听过这个词，甚至读过这方面的书。不过，现在请抛弃原有的观点，从全新的、准确的定义开始，重新认识“软件设计”。

软件是什么，我们都知道，因此真正要定义的就是“设计”了。

动词

(1) 为创造性活动制订计划。例句：工程师本月会设计一座桥梁，下个月建造它。

名词

(1) 为某种创造性活动而制订，但尚未实施的计划。例句：工程师已经确定了桥梁的设计，下个月建造它。
(2) 业已存在的创造物所遵循的计划。例句：那座桥的设计相当不错。

谈到软件设计，下列这些定义都适用。

- 我们“设计软件”（动词“设计”）时，是在进行计划活动。设计软件时要关心的事情有很多，包括代码的结构、所用的技术等，还要制订许多技术决策。通常，我们只是在脑子里做这些决定，有时候会把它写下来或画出来。
- 上一步的结果是“软件的设计”（第一种意义的名词“设计”），也就是计划。同样，它可能是落实下来的文档，也可能是我们脑中的若干决定。
- 已经存在的程序同样有“设计”（第二种意义的名词“设计”），也就是它的结构，或者它所遵循的计划。有些程序可能根本没有清楚的结构，所以是“无设计”的，也就是说开发它的程序员没有任何明确的计划。在“无设计”和“完整的设计”之间，存在着广阔的灰色地带，比如“部分的设计”、“在某段程序中存在的若干矛盾的设计”、“接近完成的设计”等等。一些刻意而为的糟糕设计甚至比无设计还要差劲。比如你遇到的那些刻意制造纠结或复杂的程序，就属于这类刻意而为的糟糕设计。

软件设计的科学就是为软件做计划、制定决策的科学，它帮助大家做出这类决定：

- 程序的代码应当采用什么结构？
- 是程序的速度重要，还是代码容易阅读重要？
- 为满足需求，应该选择哪种编程语言？

软件设计与下列问题无关：

- 公司的结构应该是怎样的？
- 什么时候召开团队会议？
- 程序员的工作时间应该如何安排？
- 程序员的绩效如何考核？

这些决策与软件本身无关，只与组织有关。显然，保证这类决策的合理性也是重要的——许多软件项目之所以失败，就是管理失当。但是这不是本书的主题，本书关注的是，如何为你的软件制订合理的技术决策。

> **软件系统中任何与架构有关的技术决策，以及在开发系统中所做的技术决策，都可以归到“软件设计”的范畴里。**

2.1 程序员也是设计师

在软件项目中，每个程序员的工作都与设计有关。首席程序员负责设计程序的总体架构；高级程序员负责大的模块；普通程序员则设计自己的那一小块，甚至只是某个文件的一部分。但是，即便仅仅是写一行代码，也包含设计的因素。

哪怕是单干，也离不开设计。有时候你在敲键盘之前，就做了决定，这就是在做设计。有的夜晚，你躺在床上，还在思考要怎样编程。

每个写代码的人都是设计师，团队里的每个人都有责任保证自己的代码有着良好的设计。任何软件项目里，任何写代码的人，在任何层面上，都不能忽略软件设计。

这不是说设计应该采取民主程序进行。设计决不应该由某个委员会负责，那样的结果必然很差劲。相反，所有的开发人员都应有权在自己的工作中做出良好的设计决策。如果某位开发人员做了糟糕或者平庸的决策，资深开发人员或者首席程序员就应当推翻这些决策，重新来过。对下属的设计，这些人应当拥有否决权[1]。不过，软件设计的责任应当落实在真正写代码的人身上。

身为设计师，必须时时愿意聆听建议和反馈，因为程序员大都比较聪明，有不错的想法。但是考虑了所有这些建议和反馈之后，**任何决策都必须由单独的个人而不是一群人来做出。**

2.2 软件设计的科学

现在，软件设计仍然不算科学。什么是科学？词典里的定义有点儿复杂，但简单来说，一门学问要成为科学，必须符合下列标准。

- 科学必须包含**汇总而来的知识**。也就是，它必须包含事实而不是意见，且这些事实必须汇总起来（比如集结成书）。
- 这些知识必须具有某种**结构**。知识必须能分类，其中的各个部分必须能够依据重要性之类的指标，妥善建立起与其他部分的联系。
- 科学必须包括**一般性的事实**或者**基本的规则**。
- 科学必须告诉你在现实世界中如何**做一些事情**。它必须能够应用到工作或生活中。
- 通常，科学是经由**科学方法**来发现或证明的。科学方法必须观察现实世界，提炼出关于现实世界的理论，通过验证理论，而且这些实验必须是可重复的。这样才能说明理论是普适的真理，而不仅仅是巧合或者特例。

注1：如果你是推翻决策的那个人，这么做的时候应当教育其他程序员。你应当说明，为什么新决策比原来的好。这样下去，你以后推翻的决策就会越来越少。有些程序员从来也不学习，即便是这样教育几个月、几年，他们还是会做出成堆的糟糕决策，这种人应该清理出团队。不过，大多数程序员都非常聪明，学得很快，所以这基本不是一个问题。

在软件领域，我们已经掌握了很多知识。软件的知识已经被著书阐述，已经有了结构。不过，除了成为科学所必需的这些条件，我们还忽略了最重要的部分：清楚表述出来的**规则**（law）。规则是恒常不变的事实，一旦掌握了它们，人就不会犯错。

有经验的软件开发人员知道怎么做才是正确的。但是为什么这么做就是正确的？判断正确和错误的标准在哪里？

软件设计的基础规则是什么？

本书将给出答案。书中列出了关于软件开发的若干定义、事实、规则、定律，它们的着眼点大多在于软件设计。可是，定义（Definition）、事实（Fact）、条例（Rule）、规则（Law）的差别在哪里呢？

- **定义**告诉你事物是什么，应当如何使用。
- **事实**是关于事物的真实陈述。每一点真实的信息都是事实。
- **条例**是给你的确切建议，它包含某些具体的信息，用于制订决策。但是，条例并不能帮你绝对准确地预测未来，也不能帮你发现其他真理。它们通常会告诉你是否需要采取某些行动。
- **规则**是永远为真的事实，它涵盖了很多领域的知识。它们帮你发现其他重要的真理，帮你预测未来要发生的事情。

所有这些里面，规则是最重要的。阅读这本书，你会清楚地认识到某些规则，因为书里会明确说明。如果你不是很确定某些信息应该如何归类，可以查看附录 B，其中列出了本书中所有重要的信息，并标注了它到底是条例、规则、定义，还是普通的事实。

看到这些定义、条例、规则、事实的时候，你可能会想："这些都是显然的，我早就知道了。"这点不用想也知道，你已经在软件开发行业里摸爬滚打很久了，大概早就遇到过这些概念。但是，你这么想的时候应该问问自己：

- 我是否知道，某些特定的说法是否经过了证实？
- 我是否清楚它的重要性？
- 我是否可以向其他人清楚讲解，让对方彻底明白？

- 我是否明白，在软件开发领域中，它与其他部分知识的关系如何？

如果对某些问题，你曾经说过“不是”、“或者”，但现在可以说“是”，你就已经有所领悟。要区分某些说法到底是科学，或者仅仅是想法的集合，这种领悟是非常重要的。

当然，科学不等于全知全能。宇宙里还存在待发现的秘密，在任何领域都存在着未知的事物。有时候，发现了新的数据或者事实，就必须修正某些基础规则。但这正是新的出发点！我们可以由此构筑科学：关于开发软件的、切合实际的且能够观察验证的规则和真理。

即便在某天，本书的某部分被证明为错的，并出现了更完善更先进的科学，但有一个事实是很清楚的：**软件设计可以成为科学。**这门学问不是什么永恒的秘密，因为程序员在不断成长，咨询顾问要把软件设计“新方法”拿出去换钱。

> **软件设计是有章（规则）可循的，它们可以被认识，可以被理解。规则是永恒不变的，是基本的事实，而且确实可行。**

至于本书中的规则是不是正确，我们可以引用几百个例子和试验来证明它，不过最终，你需要自己判断。请检验这些规则，看看你是否能想到关于软件开发的，适用范围更广或更基础、更深入的普适真理。如果你有所发现，或者找到了这些规则的问题，可以登录 http://codesimplicity.com/ 联系作者，提供你的贡献或问题。关于这门科学的任何进一步发展，都可以让所有人受益，只要新的内容是关于软件设计的真实的、基本的定律或规则。

2.3 为什么不存在软件设计科学

你可能很好奇，这本书诞生之前，为什么不存在软件设计的科学。毕竟，软件开发已经有几十年历史了。

让我来讲个有趣的故事吧。这些背景知识有助于你了解，为什么我们跟计算机打了这么久的交道，却没有建立起软件设计的科学。

今天的"计算机"来源于数学家的设计，它最初纯粹是一种抽象的设备，数学家想用机器代替人脑来解数学问题。

这些数学家才是计算机科学之父，计算机科学正是对信息处理所做的数学研究。它并不像现在一些人认为的那样，是关于计算机编程的学问。

不过，最早的计算机是在计算机科学家的指挥下、由经验丰富的电子工程师制造出来的，它由受过专门训练的操作员在严格控制的环境下操作。那时的计算机是为有需要的机构和部门（大多数是政府部门，为了计算弹道或者破解密码）量身定制的，每种型号只生产一两台设备。

然后 UNIVAC 出现了，这是世界上第一台商用计算机。当时，世界上还有足够多的资深的理论数学家。到了普通人也买得起计算机的时候，已经不可能为每台计算机配一名数学家了。有些组织机构，比如美国统计局（也就是 UNIVAC 的首批客户之一），肯定是有足够多的受过严格训练的计算机操作员的。其他机构则只能先买来机器，然后说把它交给员工："好，财务部的 Bill，这机器归你了，仔细看看操作手册，让它跑起来。"于是 Bill 埋头学习这台复杂的机器，尽可能让它跑起来。

就这样，Bill 成为了第一批"应用程序员"。他可能在学校学过数学，但是他几乎肯定没有学习过设计和运行计算机需要用到的高深理论。不过，他可以阅读手册来理解，并通过试错法让计算机照自己的想法工作。

然后，商用计算机卖得越来越多，Bill 这样的人也越来越多，而严格训练的操作员显得越来越少。大多数的程序员就像 Bill 一样。如果说 Bill 有什么不一样的地方，那就是工作压力。主管会要求他"完成这个任务"，而且会说"我们不管你怎么做，总之去做就是了"。他必须根据说明书，想象出该如何完成任务，然后完成，即便他的程序每两个小时崩溃一次。

最终，Bill 带领一组同事一起工作。他必须能够设计出系统，并且将任务分派给不同的人。于是，实践型编程的技艺不断发展完善，这个过程更像大学生的自学，而不是 NASA 的工程师建造航天飞机。

在这个阶段，软件开发就像一锅大杂烩，非常复杂，也难于管理，但是，每个人都有点儿心得。然后出现了 Fred Brooks 写的《人月神话》，探究了某个实际的软件项目的开发过程，并指出一些事实。其中最著名的就是，项目延迟的情况下，增加更多的程序员，只能加剧延迟。Brooks 并没有提出完整的科学，但是他对编程及软件开发管理的观察相当有价值。

这之后涌现出了大量的软件开发方法：Rational 统一过程、能力成熟度模型、敏捷软件开发，等等。它们无一标榜自己是科学——虽然它们只是管理软件开发复杂性的手段。

正是这一切，导致了现状：方法众多，真正的**科学**却缺席。

其实，缺席的科学分为两种：软件管理的科学，软件设计的科学。

软件管理的科学告诉我们的是，如何为程序员分派工作，如何制订发布计划，如何估量任务所需的时间，诸如此类。这是一门显学，上述的各种方法强调的正是这一点。在这个领域中，存在着彼此矛盾却各有道理的观点，这说明软件管理的基本规律尚未被认识。不过，大家已经关注到了这个问题。

但是，软件设计的科学在现实的编程中却没有什么人关注。学校里几乎没人教导你说，软件设计中存在着科学。相反，大多数人听到的都是：编程语言就是这样的，现在就动手写代码吧。

本书将要弥补这点缺憾。

本书要讲的并不是计算机科学，那属于数学研究。本书提供的是“实际干活的程序员”所需科学的入门部分——在任何语言中编写程序时都应当遵循的若干基础定律和规则。这种科学与物理和化学一样可靠，它告诉你如何编写程序。

据说，这样的科学是不存在的，软件设计的变化太多了，无法用简单、基础的规则来描述，但这只是一种说法。有些人也说过，理解物理世界是不可能的，因为“世界是神创造的，但神是不可知的”，可是我们确实已经发现了物理世界的规律。所以，除非你相信计算机是不可知的，否则，软件设计的科学是必然会存在的。

也有人说，编程纯粹是一门艺术，服从于程序员的个人爱好。这么说有点道理，在科学应用的任何领域，都存在着不少艺术的成分，但是它们背后必然有科学存在。只可惜，关于程序设计的科学至今仍然是一片空白。

其实，软件复杂性的主要根源就在于缺乏科学。如果程序员有科学指导，知道该如何开发简单的软件，就不会有这么多的复杂性问题，我们也不需要运用令人发狂的过程来管理这些复杂性。

第3章

软件设计的推动力

在编写程序时，我们应当知道自己为什么要编程，最终目标是什么。

有没有什么办法，可以把各种软件的目标都汇总到一起？如果有，那么整个软件设计的科学就有了方向，因为我们已经知道自己要做什么。

其实，全部软件都有一个相同的目标：

> **帮助其他人。**[1]

依据具体的软件，我们可以得到更加具体的目标。比如，文字处理软件帮助大家编辑文档，浏览器帮助大家浏览网页。

有些软件是帮助**特定人群**的。比如，有许多审计软件给审计师用，此类软件仅服务于这一人群。

那么，服务于动物或植物的软件呢？软件服务动物和植物的真正目的，还是帮助**人类**。

重要的是，软件从来都不是用来帮助无生命事物的。软件要帮助的不是计算机，而是人。即便你写的是程序库，它们也是为程序员服务的，程序员也是人，你的服务对象并不是计算机。

“帮助”究竟是什么意思呢？从某些角度来看，这是个主观的概念——对这个人有帮助，并不等于对那个人也有帮助。不过词典里有这个词的定义，所以它的定义并不完全依赖于个人的理解。《韦伯斯特新世界美语词典》里是这么定义“帮助”的：

> （让人）能更容易地做某事；帮忙；协助。特指完成部分工作，减轻或者分担工作量。

你可以帮很多忙——做规划、著书、制订食谱等。想干什么取决于你，但目标都是**提供帮助**。

注 1：这个目标比任何规则都重要。在英语中找不到简单的词来描述它，我们或许可以称其为“更高级规则”，虽然严格来说它还算不上“规则”（比如，它不能预测未来）。为简单起见，书末附录里把这一目标归类为规则，其他时候，我们称其为“软件的目标”。

软件的目标不是“赚钱”或者“炫耀智商”。不管是谁，只要将这些当成自己的目标，就偏离了软件的目标，就很可能惹上麻烦。即便赚钱和炫耀智商可以“帮助”你，但是这只是非常狭隘的帮助；相比为帮助别人而设计的、满足其他人需求的软件，仅仅考虑狭隘目标的设计，很可能收获糟糕的软件。[2]

不理解“帮助其他人”的程序员，只能写出糟糕的程序，也就是说，他们的程序提供不了什么帮助。实际上，大概存在这么一点理论（根据对大量程序员的长时间观察所得的猜想）：一个人写出优秀软件的**潜力**，完全取决于他在多大程度上理解了“帮助其他人”的思想。

总的来说，在做与软件有关的决策时，指导法则就是判断能提供什么样的帮助（请记住，帮助有很多种，帮大忙、帮小忙，帮很多人、帮少数人）。各项需求也可以照这个标准排出先后顺序。哪项需求能为人们提供最大的帮助，就应当赋予最高的优先级。关于需求的优先级，还有很多可以补充，但是在评估开发或维护软件系统的建议时，“它能给用户帮多少忙”绝对是一个重要而且基础的问题。

总之，在设计软件时，应当将目标——帮助他人——视为应该考虑的最重要因素，这样，我们才能认识并了解软件设计的真正科学。

真实应用

如何才能把软件的目标落实到真正的项目中去呢？举个例子吧，假设我们要为程序员开发一款文本编辑器。首先需要决定的就是软件的目的。这个目的最好要简单，不妨先定位“帮助程序员编辑文本”。更具体一点也没问题，甚至有时候会更好。如果开发人员对更具体的目标尚存分歧，且从最简单的形式开始吧。

注 2：没错，你当然可以把“赚钱”当成自己或者公司的目标，这无可厚非，但它不应该是你的软件的目标。任何情况下，你所赚的钱都直接维系于你的软件能为他人提供多少帮助。事实上，决定软件公司收入的两个主要因素基本就是组织水平（包括行政、管理、推广、销售等方面），以及软件对他人的帮助。

既然目标确定了，来看所有需要完成的功能吧。针对每一条功能，我们都要问自己："这个功能怎样帮助程序员编辑文本呢？"如果答案是没什么帮助，功能就应当马上取消。这样检查过一遍之后，对剩下的每个需求，要写下一个短句作为答案。假设，有人要求我们为常用操作提供快捷键。我们应该这么记："快捷键可以帮助程序员编辑文本，因为有了快捷键，程序员不用浪费时间输入，他们与程序交互的速度提高了。"（你不一定要写下来，如果写下来显得多余，只要保证自己明白这一点就好了。）

之所以要这样问自己，是因为还存在其他有价值的理由。

- 这样思考，有助于澄清功能描述或实现方式中的模糊之处。上面关于快捷键的描述说明，它的响应必须很快，因为快捷键的价值就在这里。
- 这样思考，有助于团队确认功能的价值。有些人可能不喜欢快捷键，但是每个人都应该认同上面关于其价值的解释。事实上，有些开发人员会想出更好的办法来满足客户的需求（进一步提高操作文本编辑器的速度），而不需要快捷键，这样也是没有问题的。如果问题的答案可以引出更好的想法，就应该去实现那个更好的想法。问题的答案告诉我们的是真实的需求，而不是用户以为的需求。
- 这样思考，某些功能会凸显出更重要的价值。这有助于项目领导分配优先级。
- 退一万步说，如果编辑器因为堆叠了过多功能而臃肿，可以根据这些答案决定删减哪些功能。

我们还需要列一张bug清单，方便检查和反思：这个bug如何影响程序员编辑文本？有时候答案很明显，那么也不需要写下来：比如你要保存文件的时候程序崩溃了，就不需要画蛇添足地解释为什么崩溃是不好的。

要把软件的目标落实到真正的工作中，还有很多办法，这里只给出了几个例子。

软件设计科学的目标

了解了软件设计的目标之后，就可以确定软件设计的科学的基本方向了。

从目标看，我们知道开发软件是为了帮助其他人。所以，软件设计科学的目标应该是：

> *确保软件能提供尽可能多的帮助。*

其次，我们通常希望软件可以给大家提供持续的帮助。所以，第二个目标是：

> *确保软件能持续提供尽可能多的帮助。*

这个目标相当伟大，但是，不管是什么软件系统，也不管规模多大，它都是非常复杂的，所以确保持续提供帮助其实是个挑战。实际上，在今天，编写和维护有帮助的软件的主要障碍在于设计和编程。如果软件很难开发或修改，程序员的主要精力就花在让软件“能用”上，而没有精力去帮助用户。如果系统易于修改，程序员就会有更多的余力去帮助用户，不必费心于编程细节。同样道理，软件的维护难度越低，程序员确保软件能持续提供帮助的难度也越低。

于是，我们得到了第三个目标：

> *设计程序员能尽可能简单地开发和维护的软件系统，这样的系统才能为用户提供尽可能多的帮助，而且能持续提供尽可能多的帮助。*

传统上认为，这第三个目标就是软件设计的目标，虽然从来也没有人这么清晰地表述过。不过，第一个和第二个目标的指引是非常重要的。我们需要记住，推动实现第三个目标的，就是确保软件现在能提供帮助，将来仍然能帮助。

关于第三个目标，需要特别指出的是“尽可能简单”的说法。它的意思是软件的开发和维护都应当简单，要避免困难或复杂。这并不是

说，任何事从一开始就要简单，有时候，学习新技术，做一份好的设计，都要花很长时间，但是长期来看，你做出的选择必须确保开发和维护软件都比较简单。

有时候，第一个目标（确保有帮助）和第三个目标（维护简单）是有点冲突的，为了确保有帮助，维护难度就会增加。不过纵观历史，这两个目标冲突的原因往往并不是如此。开发出完全可维护、对用户极为有用的软件，有绝对的事实依据。其实，如果你不能确保软件的可维护性，就很难实现第二个目标，也就不能持续提供帮助。这样看来，第三个目标是非常重要的，否则我们就没办法完成前两个目标。

第4章

未来



软件设计师面对的主要问题是："在设计软件时，应该做怎样的决定？"面对的众多可能，哪一个才是最好的。我们不是要确定**绝对的好坏**，而是要知道："这些可能的选择中，哪些更好？"这是个排序问题，我们要做的是从所有可能中选出最好的决定。举个例子，设计师可能问自己："摆在眼前的功能有 100 项，但我们的人力只能够完成 2 项。应该选哪 2 项呢？"

4.1 软件设计的方程式

上面的问题，还包括软件设计的本质中的所有问题，都可以用下面的方程式来解答：

$$D = \frac{V}{E}$$

其中：

D 表示这个变化的合意程度（可取程度）。我们对此项工作的需求有多么迫切？

V 表示它的价值。该变化价值几何？一般来说，你可以问自己"这个变化对用户有多少用"；当然，还有很多其他方法来判断其价值。

E 表示完成这个变化的成本，也就是完成它需要付出的代价。

所以，这个等式的意思是：

> **任何一点改变，其合意程度与其价值成正比，与所付出的成本成反比。**

这并不是在判断某个变化绝对对错，而是指导你如何分辨并排序你的选项。能带来较大价值、花费成本较少的变化，要比带来较少价值、花费较多成本的变化"更好"。

哪怕你的问题是"我们是否应当维持不变"，也可以通过这个方程式得到答案。你可以问自己"维持不变的价值何在"或者"维持不变需要花什么成本"，再对比进行改变的价值和成本的比率。

计算机带来了社会的剧变。因为有了它，我们可以用更少的人，干更多的事情。这就是计算机的价值——它可以干很多的事情，而且速度相当快。

这挺棒的。

但是，计算机会出问题，而且总出问题。如果你家里其他东西出问题有计算机那么频繁，你多半会退货。生活在现代的大多数人，每天至少会遇到一次系统崩溃或者程序错误。

这就没那么棒了。

1.1 计算机出了什么问题？

为什么计算机这么容易出问题？如果是软件的问题，那么有而且只有一个原因：程序写得太糟糕。有些人怪罪管理，有些人怪罪客户，但是调查发现，问题的根源通常都在于编程。

那么，“程序写得太糟糕”是什么意思？这是个很模糊的说法。通常来说，程序员都是很聪明、很理智的人，他们怎么会编出“糟糕”的程序呢？

说穿了，这一切都与复杂性有关。

今天，计算机大概是我们能生产的最复杂的设备了。它每秒钟可以计算数十亿次，它内部数以亿计的电子元件必须精密协调，整台计算机才可以正常运行。

计算机上跑着的程序同样复杂。举例来说，微软的 Windows 2000 还在开发时，就可算有史以来规模最大的软件了，它包含 3000 万行代码，这大概相当于 2 亿字——是《不列颠百科全书》字数的 5 倍。

程序的复杂性问题可能更加麻烦，因为程序里没有摸得着的东西。程序出问题的时候，你也找不到什么实实在在的东西，打开瞧瞧里面发生了什么。程序完全是抽象的，非常难处理。其实，常见的计算机程

4.1.1 价值

方程式中的“价值”指什么？价值最简单的定义是：

> 这个变化能带给人多大帮助。

所有需要帮助的人里面，最重要的是你的用户。不过，为了赚更多钱而开发某些功能，也是一种形式的价值——对你来说有价值。其实，创造价值的方式有很多种，上面只是举两个例子。

要准确衡量某点变化所创造的价值，有时候是很难得。如果你的软件可以帮人减肥，你怎么精确衡量帮人减肥的价值？这是做不到的。不过你可以知道的是，软件的某些功能可以帮人减掉不少体重，有些功能则对减肥完全无效。到头来，你还是可以根据价值来为变化排序。

判断每一点可能的变化的价值，基本的依据是开发人员的开发经验，还包括对用户做恰当的研究，找到对他们帮助最大的工作。

1. 价值可能性和潜在价值

价值由两部分组成：可能价值（这个变化有多大可能帮到用户）、潜在价值（这个变化在对用户提供帮助的时候，将为用户提供多大的帮助）。

举例来说：

- 某个功能可以延长人的寿命，即便只有百万分之一的可能性，这也仍然是非常有用的功能。它的潜在价值很高（延寿），可能价值非常低。
  再举个例子，你可以为电子表格程序添加功能，以便盲人输入数字。盲人只占用户的很小一部分，但如果没有这个功能，他们就根本没法使用你的程序。同样，这个功能是有价值的，因为其潜在价值很高，尽管它只对一小部分用户有用（可能价值很低）。
- 能够让所有用户都微笑的功能确实是有价值的。其潜在价值很低（只能让人微笑），但是它能影响到很多人，所以它的可能价值非常高。

- 反过来，如果你完成的某项功能，只有百万分之一的可能性让某人微笑，那么它并没多少价值。这项功能的潜在价值和可能价值都很低。

所以，在判断价值时，你应该考虑：

- 多少用户（占多大比例）会从此项工作中受益？
- 此功能对用户有价值的可能性有多大？或者换个说法：此功能发挥价值的频率有多高？
- 在它发挥价值的时候，它能发挥出多大的价值？

2. 平衡危害

有些改变在带来帮助的同时也会带来麻烦。如果你的程序要展示广告，就会让某些人觉得烦，虽然展示广告可以为开发提供资金。

权衡改变的价值，需要衡量它可能造成的危害，并权衡利弊。

3. 赢得用户的价值

如果某个功能找不到用户，就不存在实际的价值。这些功能可能用户找不到，或者很难使用，或者根本就帮不上任何人。在将来它们可能有价值，但目前没有。

这也意味着，大多数情况下，为了确保你的软件有价值，就必须真正发布它。如果某个变化花费的时间太长，最后可能就没有任何价值，因为它不能及时发布，无法给人提供切实的帮助。在判断某项工作的合意程度时，评估其发布计划是非常重要的。

4.1.2 成本

比起价值，成本更容易量化一点。通常，你可以计算“需要多少个人工作多少小时”。经常听到的关于成本的量化表述是“100 人年”，它表示 1 个人工作 100 年，或者 100 个人工作 1 年，或者 50 个人工作 2 年。

尽管成本可以量化，落实起来却相当复杂，甚至是不可能的。有些变

化包含隐形成本，难于预测，比如花在修正开发某项功能造成的bug上的时间就很难预测。不过，经验丰富的软件开发人员仍然可以在不需要知道确切数值的情况下，根据可能成本来预测。

预测某个变化的成本时，重要的是要考虑牵涉到的所有投入，而不仅仅是编程的时间。研究要花多少时间？开发人员之间的沟通要花多少时间？所做的思考要花多少时间？

简单说，与这个变化有关的每一点时间，都是成本的一部分。

4.1.3 维护

到目前为止，这个方程式还非常简单，但它忽略了一个重要的因素——时间。你不但需要完成这个变化，还需要一直维护它。所有的变化都需要维护。有些工作显而易见要做维护：如果你开发一款计算报税的软件，每年的新规定公布之后，你都必须升级。即使这项工作暂时看不到长期的维护成本，即使维护成本只是你下一年必须再做一次测试，维护成本仍然存在。

我们还必须考虑现在价值和未来价值。修改现有的系统时，受益的是现在的用户，但是它也可能帮助到未来的用户，甚至可能影响到未来的用户数目，进而影响到现有系统究竟总共能帮助到多少人。

有些功能的价值会随时间变化。举例来说，2009 年的税收计算软件在 2009 年和 2010 年是有价值的，但到了 2011 年就要打折扣了。这就是随时间变化而贬值的功能。还有些功能会随时间变化而增值。

所以，脚踏实地来看这个问题，我们会发现，成本包含**实现成本**和**维护成本**，价值也包括**当前价值**和**未来价值**。用方程式来表示就是：

$$E = E_i + E_m$$
$$V = V_n + V_f$$

其中：

E_i 代表实现成本

E_m 代表维护成本
V_n 代表当前价值
V_f 代表未来价值

4.1.4 完整的方程式

考虑所有因素，完整的方程式就是：

$$D = \frac{V_n + V_f}{E_i + E_m}$$

用文字说明就是：

> 改变的合意程度（可行性），正比于软件当前价值与未来价值之和，反比于实现成本和维护成本之和。

这就是软件设计的最重要规律。不过，还需要做一些补充说明。

4.1.5 化简方程式

“未来价值”和“维护成本”都取决于时间，如果把这个方程式应用到现实中，随着时间的改变，会出现非常有意思的现象。为说明这个问题，假设价值和成本都可以用金钱来衡量。“价值”衡量的是这项工作能带来多少钱，“成本”衡量的是为完成这项工作需要投入多少钱。在真实世界中不应当这样处理，但就这个例子而言，这么做可以起到简化作用。

好，假设要完成工作对应的方程式是这样：

$$D = \frac{\$10\,000 + \$10\,000/\text{天}}{\$1000 + \$100/\text{天}}$$

这项工作要花 1000 美元完成（实现成本，分母的第 1 项），能马上带来 10 000 美元的收益（当前价值，分子的第 1 项）。之后每一天，它都能带来 1000 美元收益（未来价值，分子的第 2 项），而且需要花 100 美元来维护（维护成本，分母的第 2 项）。

到第 10 天，累积的未来价值是 10 000 美元，累积的维护成本是 1000

美元。这也相当于之前说的“当前价值”和维护成本，只是时间在 10 天后而已。

在 100 天后，未来价值一共是 100 000 美元，维护成本是 10 000 美元。

在 1000 天后，未来收益总计为 100 万美元，维护成本总计为 10 万美元。这时候，最开始说的“当前价值”和维护成本看来就微不足道了。随着时间的流逝，它会越来越不重要，甚至完全无足轻重。于是，随着时间流逝，这个方程式变成了[1]：

$$D = \frac{V_f}{E_m}$$

其实，几乎所有软件设计的决策都完全忽略了未来价值与维护成本的对比。有时候当前价值和实现成本足够大，始终在决策中占有决定性地位，但是这种情况很少见。一般来说，软件系统都需要维护很长时间，大多数情况下，未来长期收益和维护成本才是真正需要考虑的，与之相比，当前价值和实现成本变得无足轻重。

4.1.6 你需要什么，不需要什么

我们已经学到了重要的一课，也就是要避免这样的情况：对某项工作，维护成本最终大大超过未来的收益。假设你完成某项工作，在 5 天内，它的成本和价值看起来是这样的：

天数	成本	价值
1	$10	$1000
2	$100	$100
3	$1000	$10
4	$10 000	$1
5	$100 000	$0.10
总计	$111 110	$1111.10

显然，这是一项非常糟糕的工作，千万不要这么干。如果按照这个势

注 1：补充说明：如果学过微积分，你可能会认识到，我们正在分析这个等式在时间接近无穷时的极限。通常来说，你应该把软件设计的方程式理解为一个具有极限的无穷数列，而不是某个静态方程式。不过，为简单起见，现在先按静态方程式来考虑。

头持续下去，系统根本无法维护——它会变得无比昂贵，而每天都没有产生收益。

无论如何，只要维护成本的增长速度比价值快，你就会遇到麻烦，即便刚开始的情况是这样的：

天数	成本	价值
1	$1000	$1000
2	$2000	$2000
3	$4000	$3000
4	$8000	$4000
总计	$15 000	$10 000

理想的解决方案——也即保证成功的唯一途径——就是这样设计你的系统：保证维护成本随时间降低，最终降到零（或者尽可能接近零）。只要你能做到这点，就无所谓未来收益是大还是小，总之你不需要关心。比如，下表所示就是合意的情况。

天数	成本	价值
1	$1000	$0
2	$100	$10
3	$10	$100
4	$0	$1000
5	$0	$10 000
总计	$1110	$11 110

天数	成本	价值
1	$20	$10
2	$10	$10
3	$5	$10
4	$1	$10
5	$0	$10
总计	$36	$50

能带来更高未来价值的改变仍然是更可取的，不过，只要每项决策的维护成本都会随时间降低到接近于零，未来你就不会陷入危险的境地。

理想情况下，只要未来收益高于维护成本，工作就是值得做的。所以，哪怕维护成本和未来收益都增加，只要未来收益超过维护成本，也是值得做的。

天数	成本	价值
1	$1	$0
2	$2	$2
3	$3	$4
4	$4	$6
5	$5	$8
总计	$15	$20

这样的工作也不错，但是更合意的是这样的工作：虽然实现成本很高，维护成本却会下降。如果维护成本会随时间推移而逐步降低，这项工作就会变得越来越有价值。所以我们才说这样做更合意。

通常来说，如果要设计一个系统，其维护成本能逐渐降低，需要在实现上花很高的成本，做相当多的设计和规划。不过请记住，在设计决策中，实现成本通常并不是重要的因素，所以基本可以忽略。

简而言之：

> **相比降低实现成本，降低维护成本更加重要。**

在软件设计的科学中，这是非常重要的一点。

但是维护成本从哪里来？怎样设计系统，才能保证维护成本会随时间推移而降低？本书剩下的篇幅，大部分讲的都是这个主题。但是在开始之前，先要对未来做更细致的检查。

4.2 设计的质量

如今，要写出一款能帮助某个人的软件，实在是太容易了。但是，要写出能帮助几百万人，而且是在未来几十年里持续提供这种帮助的软件，则要难得多。那么，编程中的大部分精力应该花在哪里？大多数人会在什么时候使用这款软件？现在，还是未来的几十年？

答案是，相比现在，将来有多得多的编程工作要做，也有更多的用户要帮助。在未来，你的软件必须保持竞争力，才能继续存在，同时，维护成本和用户数量也会增加。

如果我们忽视事实，放弃对未来的思考，只考虑当下"能用"的软件，那么我们的软件在未来就会很难维护。如果软件很难维护，就很难确保它能够帮助别人（而这正是软件设计的目标）。如果你不能添加新功能，也不能修正问题，你最后得到的就是"糟糕的软件"。它不能帮到用户，而且满身问题。

于是，可以得到这条规律：

> **设计的质量好坏，正比于该系统在未来能持续帮助他人时间的长度。**

如果你的软件只能在未来几小时内提供帮助，就不需要花太多的工夫去设计。如果需要在未来 10 年内都能派上用场（这个时间通常会超过你的预期，虽然你很可能只打算保证能用 6 个月），就需要花很多精力去设计。如果举棋不定，不妨设想它要使用很长很长的时间，然后按照这个设想去设计。不要把自己禁锢在某种工作定势里，要保持灵活；不要做任何以后无法改变的决策；在设计时要慎重，慎重，再慎重。

4.3 不可预测的结果

现在我们知道了，设计软件时最应该关注的是未来。不过，关于任何工程，都有一点极为重要：

> **未来的某些事情，是我们所不知道的。**

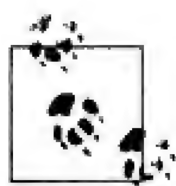
其实，软件设计也是如此，关于未来，大多数事情都是未知的。

> **程序员犯的最常见也是最严重的错误，就是在其实不知道未来的时候去预测未来。**

假设，1985 年有个程序员编了个修理软盘错误的程序。这个程序不能修理其他任何东西——因为它的每个部分都是专门针对软盘的工作原

理和状态而考虑的。到现在，这个软件肯定没用了，因为已经没有人用软盘了。程序员的预期是“人们会一直使用软盘”——但其实他并不确切知道这一点。

预测短期未来是可能的，但长期未来基本是未知的。可是，相比短期，长期的未来对我们来说更重要，因为我们的设计决策会在未来更长的时间里产生更大的影响。



如果完全不考虑未来，只根据当前已知的确切信息确定所有设计决策，那就百分百安全了。

这个说法听起来与本章之前说的相矛盾，其实并非如此。在进行决策时，未来才是最重要的事情。但在进行决策时，考虑未来的变数和尝试预测未来，是有区别的。

打个比方，你要做个简单的抉择，是吃东西还是饿死。这并不需要预测未来才可以决策——你当然知道吃东西是更好的决策。为什么？因为这样你可以活下去，而且将来可以过得更好，饿死了就没有未来可言了。未来是重要的，在做决策时需要考虑它。选择吃饭，是因为它可以带来更好的未来。但是，未来并不需要预测——我们不需要知道确切的未来，比如“我要吃东西，因为明天我得救一个孩子”。只要你选择吃饭，而不选择饿死，无论明天发生什么，都比饿死要好。

同样道理，在软件设计时，可以根据已知的信息做某些决策，目的是为了创造更好的未来（提升价值，降低维护成本），而不必预测未来究竟会发生什么具体的事情。

当然，也有少量例外，有时候你确切知道未来短期内会发生什么，便可以据此决策。如果这么做，必须对未来有相当的把握，而且这种未来必须触手可及。无论你多聪明，肯定没有简单的办法准确预测长期未来。

举个编程之外的例子。CD 诞生于 1979 年，最初设想是取代磁带，成为最主要的听音媒质。谁能预计到 20 年之后会出现同样尺寸的 DVD，

所以计算机上会出现同时兼容 CD/DVD 的驱动器？谁又可以预计到，当 CD-ROM 以 50 倍速读取 CD 时，会出现什么问题？

正是因为如此，任何工程——包括软件开发——都有“指导原则”。如果我们遵循这些原则，无论未来发生什么，一切事情都会按部就班。这就是软件设计的规律和法则，也是我们这些设计师的“指导原则”。

是的，重要的是要记住，存在着未来。但是，这不要求你必须预测未来，它只是说明你为什么应当遵循本书的规则和条例来决策——因为无论未来会发生什么，它们总可以保证你的软件不偏离正轨。

事实上，某条特定的规则和条例在未来会以什么样的方式帮助你也无法预测——但是它确实能帮上忙，而且你会欣然乐意把它们应用在工作中。

你当然有权不同意本书中读到的规则。关于规则，你一定要有自己的看法。但是你应当谨记，如果不遵守这些规则，你很可能在无法预测的未来落得一团糟。

序就已经足够复杂，没有人可以从头到尾理解所有代码是如何工作的。程序越大，越是如此。

这样说来，编程就成了把复杂问题化解为简单问题的劳动。否则，一旦程序达到某种复杂程度，就没有人可以理解了。程序中复杂的部分必须以某种简单方式组织起来，这样，不需要神那样强大的思维，普通程序员也可以开发出来。

这就是编程所要用到的艺术和才能——化繁为简。

“差程序员”是不会化繁未简的。他们总以为用编程语言（这东西本来就够复杂了）写出“能跑通”的程序，就已经化解了复杂性，而没有考虑降低其他程序员需要面对的复杂性。

大概就是这么一回事。

设想一下，为了把钉子钉到地板上，工程师发明了一台机器，上面有皮带轮，有绳子，还有磁铁。你多半会觉得这很荒唐。

再设想，有人告诉你说：“我需要一段代码，它能用在任何程序的任何地方，能够通过任何介质，实现两台计算机之间的通信。”这个问题无疑很难化繁为简。所以，有些程序员（大多数程序员）在这种情况下给出的解法，就像是一台装备了皮带轮、绳子、磁铁的机器，其他人当然很难看得懂。并不是这些程序员缺少理性，他们的脑子也没有进水。他们面对的问题确实困难，设定的期限也很紧，他们能做的就是这些。在这些程序员看来，写出来的东西是能用的，它符合要求。这就是他们的老板需要的，看来也应该是客户需要的。

不过，这些程序员毕竟没做到化繁为简。完工之后，他们把结果交给其他程序员，其他程序员又会在这之上继续增添复杂性，完成自己的工作。程序员对化解复杂性考虑得越少，程序就越难懂。

于是程序变得无比复杂，最终没办法找出其中的各种问题。喷气式飞机差不多也有这么复杂，但它们的造价是几百万甚至几十亿美元，而且仔细排查过错误。大多数软件的售价只有 50 ~ 100 美元。价钱这

第5章

变化



现在我们已经知道了未来的重要性，也知道了关于未来有些东西是我们不知道也不可能知道的，那么，有什么是确定的呢？

可以确定的一点是，随着时间的流逝，软件所处的环境会变化。没有东西可以永恒不变。也就是说，软件必须随环境变化而变化，才能适应所处的环境。

于是，我们得到了变化定律（Law of Change）：

> **程序存在的时间越久，它的某个部分需要变化的可能性就越高。**

未来是无穷无尽的，所以可以百分百地肯定，最终，程序的每个部分都必须变化。在未来5分钟里，可能没有哪个部分需要变化；在未来10天里，某个小部分可能需要变化；在未来20年，即便不是全部都必须变化，很可能大部分也必须变化。

哪些部分会发生变化很难预测，为什么要变化也很难预测。很可能你的程序是为4轮轿车写的，可未来每个人都开着18轮的大卡车；很可能你的程序是写给高中生的，但是高中教育的质量会差到没有学生懂你的程序。

关键在于，你并不需要去预测什么会变化，你需要知道的是，变化必然会发生。程序应该保证尽可能合理的灵活性，这样，不管未来发生什么变化，都可以应付得了。

5.1 真实世界中程序的变化

来看一份关于真正的程序随时间流逝而变化的数据吧。某个程序包含数百个文件，但是一页篇幅列不完所有的文件，所以这里选取了4个文件作样本。表5-1列出了它们的变化详情。

么低，没有人有足够的时间和资源，在几乎无限复杂的系统里找到所有的问题。

所以，“好程序员”应当竭尽全力，把程序写得让其他程序员容易理解。因为他写的东西都很好懂，所以要找出 bug 是相当容易的。

这个关于简单性的想法有时被误解为，程序不应当包含太多代码，或者是不应当使用先进技术。这么想是不对的。有时候，大量的代码也可以带来简单，只不过增加了阅读和编写的工作量而已，这是完全正常的。你只要保证，那些大段的代码提供了化解复杂性所必须的简短注释，就足够了。同样，通常来说，更先进的技术只会让事情更简单，只是一开始你得学习，所以整个过程可能没那么简单。

有些人相信，把程序写得简单所花的时间，要比写“能用就好”的程序更多。其实，花更多的时间把程序写简单，相比一开始随意拼凑些代码再花大量的时间去理解，要快得多。这个问题说起来轻巧，显得轻描淡写，其实软件开发的历史教训很多，诸多事例已经反复证明了这一点。许多大型程序的开发之所以会停滞数年，就是因为一开始没有做好，结果必须等上这么长的时间，才能给之前开发出来的怪物加上新功能。

正因为如此，计算机经常出问题——因为大多数程序都有这个问题，许多程序员在写程序时并没有化解复杂性。是的，这么做很难。但是如果程序员做不到这一点，设计出的系统过于复杂、经常出问题，用户在使用的过程中就会经受无穷无尽的折磨。这么一比，把程序写简单所费的工夫实在算不了什么。

1.2 程序究竟是什么？

大多数人说的“计算机程序”，其实有完全不同的定义：

(1) 给计算机的一系列指令
(2) 计算机依据指令进行的操作

第一种定义是程序员写程序时所用的。第二种定义是使用程序的普通用户所用的。程序员命令计算机：在屏幕上显示一头猪。这就是第一种定义，它包含若干指令。计算机接收到指令之后，会控制电信号，在屏幕上显示一头猪。这是后一种定义，即计算机执行的操作。程序员和用户都会说自己在和“计算机程序”打交道，但是他们的用法是很不一样的。程序员面对的是字母和符号，用户看到的是最终结果——计算机执行的操作。

所以，计算机程序其实是这两者的混合体：程序员的指令、计算机执行的操作。编写指令的最终结果就是让计算机执行那些操作——如果不需要执行操作，就没必要去写代码了。这就好像在生活里，你列了一张购物单（相当于指令），告诉自己该买哪些东西。如果你只是列了单子，但没去商店，单子就没有任何意义。指令必须得到实际的结果。

但是，列购物单和写程序有显著的区别。如果购物单列得很乱，只不过会降低买东西的速度。但是如果程序写得很乱，实现最终的目标就显得尤其困难。为什么呢？因为购物单是简单短小的，用完就可以扔掉。而程序是很复杂很庞大的，你可能还需要维护很多年。所以，同样是没有秩序，购物单只会给你造成一点儿小麻烦，程序却可以给你增添无尽的烦恼。

而且，除软件开发之外，没有任何领域的指令和结果联系得这么紧密。在其他领域，人们先编写指令，然后交给其他人，指令通常要等很长的时间才会执行。比如设计房子，建筑师首先给出指令——也就是蓝图。这份蓝图经很多人的手，过了很长的时间，才能建起真正的房子。所以，房子是大家所有人解读建筑师指令的结果。相反，写程序时，在我们和计算机之间没有任何人。我们让计算机干什么，就会得到怎样的结果；计算机绝对服从命令。结果的质量完全取决于机器的质量、我们想法的质量，代码的质量。

在这三个因素当中，代码的质量是如今软件工程需要面对的最重要问题。根据这一点，以及上面提到的其他原因，本书的大部分内容都在论述如何提高代码质量。会有一些地方涉及机器的质量和想法的质量，

但是大多数内容都在论述怎样改善你交给机器的指令的结构和质量。

虽然花了这么多的时间来谈代码，但也很容易忘记，改善代码质量的一切努力都是为了获得更好的结果。本书绝不姑息任何差劲的结果——我们学习提高代码质量的全部原因都在于，要想改进结果，提高代码质量是最重要的问题。

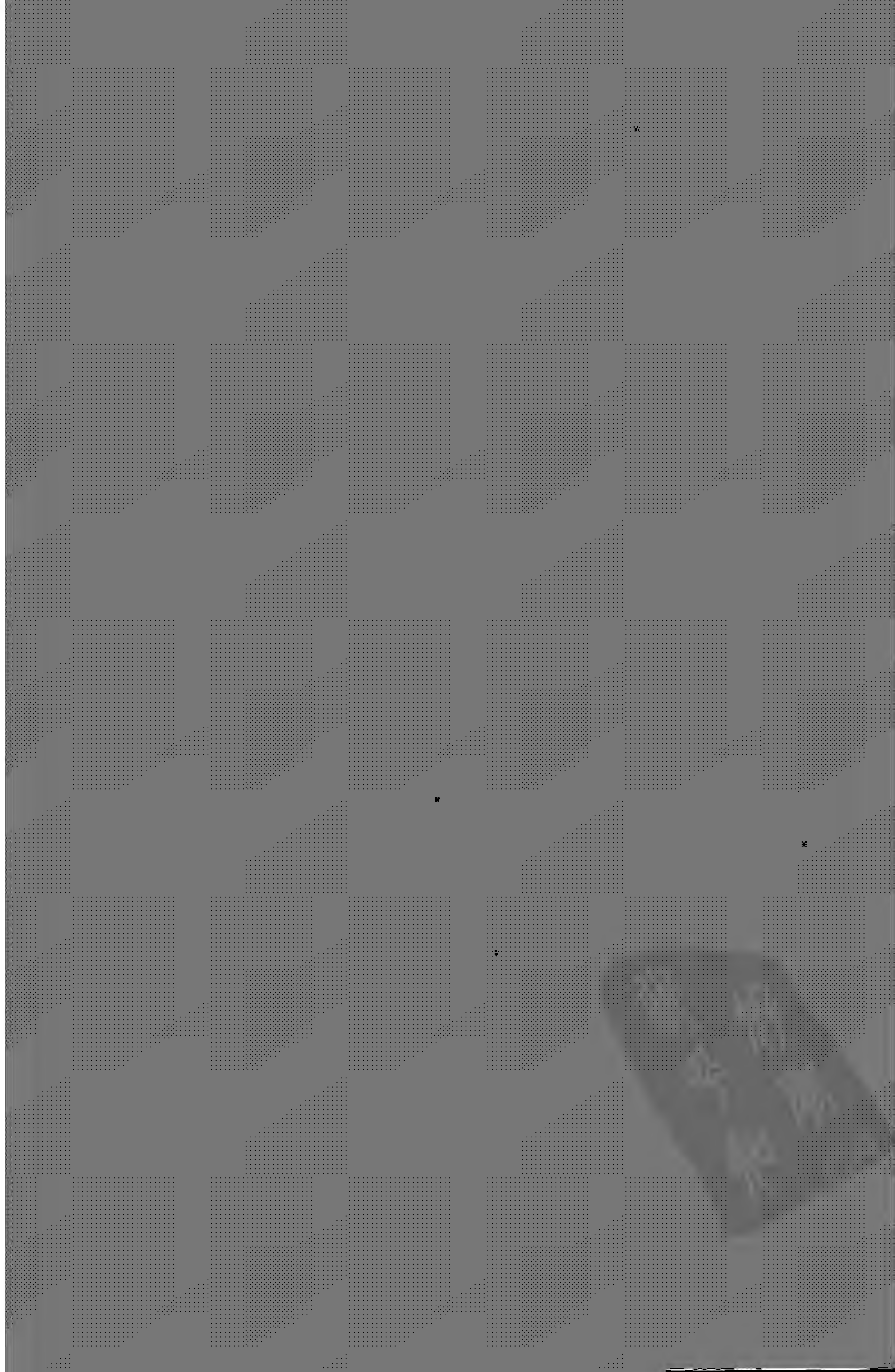
所以，我们最需要掌握的，就是提高代码质量的科学方法。



第2章

缺失的科学





附录B

事实、规则、条例、定义



本附录列出了本书所涵盖的各条重要的事实、规则、条例、定义。

- 事实：好程序员和差程序员的差别就在于理解能力。差劲的程序员不理解自己做的事情，优秀的程序员则相反。
- 条例："好程序员"应当竭尽全力，把程序写得让其他程序员容易理解。
- 定义：程序就是
  (1) 给计算机的一系列指令；
  (2) 计算机依据指令进行的操作；
- 定义：软件系统中任何与架构有关的技术决策，以及在开发系统中所做的技术决策，都可以归到"软件设计"的范畴里。
- 事实：每个写代码的人都是设计师。
- 事实：设计与民主无关，它应当由个人完成。
- 事实：软件设计是有章（规则）可循的，它们可以被认识，可以被理解。规则是恒久不变的，是基本的事实，而且确实可行。
- 规则：软件的目的就是帮助其他人。
- 事实：软件设计的目的如下。
  (1) 确保软件能提供尽可能多的帮助；
  (2) 确保软件能持续提供尽可能多的帮助；
  (3) 设计程序员能尽可能简单地开发和维护的软件系统，这样的系统才能为用户提供尽可能多的帮助，而且能持续提供尽可能多的帮助。
- 法则：软件设计的方程式

$$D = \frac{V_n + V_f}{E_i + E_m}$$

这是软件设计的主要规则，也可以这么表达：

改变的合意程度（可行性），正比于软件当前价值与未来价值之和，反比于实现成本和维护成本之和。

随着时间的推移，这个方程式会简化为：

$$D = \frac{V_f}{E_m}$$

这说明，降低维护成本比降低开发成本更重要。

- 条例：要做多少设计，应当正比于未来软件能够持续为人们提供帮助的时间的长度。
- 条例：未来的某些事情，是我们所不知道的。
- 事实：程序员犯的最常见也是最严重的错误，就是在其实不知道未来的时候去预测未来。
- 条例：最安全的情况是，完全不尝试预测未来，所有的设计决策都应当根据当前确切知道的信息来做。
- 条例：变化定律：程序存在的时间越久，它的某个部分需要变化的可能性越大。
- 事实：在落实变化法则时，软件设计师容易犯的三个错误（也就是本书中的“三大缺陷”）是：

  (1) 编写不必要的代码

  (2) 代码难以修改

  (3) 过分追求通用
- 条例：直到真正要用了才编写代码，清理掉用不到的代码。
- 条例：代码的设计基础，应当是目前所知的信息，而不是你认为未来要发生的情况。
- 事实：如果设计让事情更复杂，而不是更简单，就犯了过度工程的错误。
- 条例：在考虑通用时，只需要考虑当前的通用需求。
- 条例：采用渐进式开发和设计，可以避免三大缺陷 。
- 条例：缺陷概率法则：在程序新增缺陷的可能性与代码修改量成正比。
- 条例：最好的设计，就是能适应外界尽可能多的变化，而软件自身的变化要尽可能少。
- 条例：永远不要“修正”任何东西，除非它真的有问题，而且有证据表明问题确实存在。
- 条例：理想情况下，任何系统里的任何信息，都应当只存在一次。
- 规则：简洁定律：软件任何一部分的维护难度，反比于该部分的简洁程度。
- 事实：简洁是相对的。
- 条例：如果你真的希望成功，最好是把产品简化到傻子也能懂。

- 条例：要保持一致。
- 条例：代码可读性主要取决于字母和符号之间的空白排布。
- 条例：名字应当足够长，能够完整表达其意义或描述其功能，但不能太长，以免影响阅读。
- 条例：代码应当解释程序为什么这么做，而不是它在做什么。
- 条例：简洁离不开设计。
- 条例：你可以这样增加复杂性：
  - 扩展软件的用途；
  - 新增程序员；
  - 做无谓的改变；
  - 困于糟糕的技术；
  - 理解错误；
  - 糟糕的设计或者不做设计；
  - 重新发明轮子；
  - 背离软件原来的用途；
- 条例：可以通过考察生存潜力、互通性、对品质的重视，判断某种技术是否“糟糕”。
- 条例：通常，如果某件事情变得非常复杂，也就意味着深藏在表面的复杂之下，设计出了问题。
- 条例：在复杂性面前，问问自己“真正要解决的问题是什么”。
- 条例：大多数麻烦的设计问题，都可以用在纸上画图或写出来的办法找到答案。
- 条例：要应付系统中的复杂性，可以将系统分解成独立的小部分，逐步重新设计。
- 事实：所有可行的简化，其核心问题都是：怎么做，才可以让事情处理或是理解起来更容易。
- 条例：如果遇到不可解决的复杂性，在程序外面妥善包装上一层，让其他程序员更容易使用和理解。
- 条例：推倒重来只有在一些非常有限的情况下才是可以接受的。
- 规则：测试定律：你对软件行为的了解程度，等于你真正测试它的程度。
- 条例：除非亲自测试过，否则你不知道软件是否能正常运行。

欢迎加入

图灵社区 ituring.com.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

表5-1 随时间流逝的文件变化

	文件1	文件2	文件3	文件4
分析时常	5 年 2 个月	8 年 3 个月	13 年 3 个月	13 年 4 个月
初始行数	423	192	227	309
未变化行数	271	101	4	8
当前行数	644	948	388	414
增长数	241	756	161	105
变化次数	47	99	194	459
新增行数	396	1026	913	3838
删除行数	155	270	752	3723
修改行数	124	413	1382	3556
总变化行数	675	1709	3047	11107
变化率	1.6x	8.9x	13x	36x

表中各项指标解释如下。

分析时长：文件存在的时间长度。
初始行数：文件最初包含多少行。
未变化行数：与最初相比，没有变化的行数。
当前行数：该分析周期结束时，文件的行数。
变化数："当前行数"与"最初行数"之间的差额。
变化次数：程序员修改文件的次数（一次更改可以涉及多行）。通常一次对应着一次 bug 修正或者添加一项新功能等。
新增行数：总的来看，文件累计新增了多少行。
删除行数：总的来看，文件累计删除了多少行。
修改行数：总的来看，文件累计修改了多少行。
总变化行数：新增行数、删除行数、修改行数的总和。
变化率："总变化行数"与"最初行数"之间的比率。

上面所说的"行数"，包含文件的每一行：代码、注释、文档、空行。如果忽略注释、文档、空行，就会发现一处显著差别："未变化行数"会比其他几个指标少得多（也就是说，未变化的行几乎都是注释、文档、空行）。

看这张表格，应当弄清楚的最重要一点是，在一个软件项目中，许多变化会发生。随着时间的流逝，代码的每一行都需要修改的概率越来越大；然而，你并不能精确预测要修改什么，什么时候要修改，或者要修改多少。这 4 个文件的变化方式各不相同（只看数字也可以发现这点），但是它们都发生了相当大的变化。

关于这些指标，还有一些有趣的补充。

- 观察变化率可以发现，修改文件的工作量比最初写文件时的还要大。当然，行数这个指标不能准确反映工作量，但是它可以体现一般趋势。有时候变化率很高，比如，文件 4 的总变化行数是初始行数的 36 倍。
- 每个文件中的未变化行数，比起“初始行数”都是相当小的，比起“当前行数”来就更小了。
- 文件随时间逐渐增长只是一方面，还有很多其他变化。比如，文件 3 在 13 年后只有 161 行，但是总变化行数却有 3047。
- 总变化行数通常比当前行数要大。也就是说，如果文件已经存在了足够长的时间，你通常会去做修改，而不是新增代码。
- 观察文件 3 的指标可以发现，修改行数要多于原始行数与新增行数之和。相比新增一行，修改已有某行的频率更高。也就是说，文件中有一些行是反复修改的。如果项目持续的时间很长，这是常见的情况。

以上列出的只是部分的信息，关于这些数字，还可以做其他许多有趣的分析。我们鼓励你去探究这些数据（或者是分析自己项目里的类似指标）来取得进一步的收获。



另一点值得学习的有趣之处是，回顾某个特定文件修改历史。如果某个文件存在了很长时间，而且你有程序记录每个文件的修改历史，请回顾整个过程中的每次修改。问问自己，最初写这个文件时，你能预测到这些变化吗，是否一开始写好就能够减轻后期的工作量。总的来说，就是要尝试理解每次修改，看看是否能从中得到一些关于软件开发的新的收获。

5.2 软件设计的三大误区

为了适应变化定律，软件设计师常常会掉进误区。其中有 3 个误区最常见，这里按照其发生频率逐一列出来：

(1) 编写不必要的代码
(2) 代码难以修改
(3) 过分追求通用

5.2.1 编写不必要的代码

如今，软件设计中有一条常见的规则，叫做“你不会需要它”（You Ain’t Gonna Need It），或者简称为 YAGNI。其实这条规则的意思是，不应该在真正的需求来临之前编写那些代码。这条规则相当不错，但名字取错了。将来你可能真的需要这些代码，但是既然不能预测未来，就不知道现在该怎么使用这些代码。如果现在就编写这些代码，却不知道该怎么用，等到你真的需要使用时，就得重新设计。所以应当做的是，省下重新设计的时间，等到真正需要的时候再编写那些代码。

在需求来临之前就编写代码的另一点危险是，当前用不到的代码很可能会导致“劣化”[1]。因为代码从来没有用到，它很可能与系统的其他部分脱节，继而产生 bug，可是你永远也不知道。最终等你想去使用的时候，就得花时间去排错。更糟糕的是，你很可能相信这些从来没用过的代码是正确的而忽略了检查，这样 bug 就留给了用户。其实，这条规则应当这样展开：

不要编写不是必需的代码，并且要删除没有用到的代码。

也就是说，你还需要删除所有用不到的代码；如果真的需要，你随时可以恢复回来。

注 1：bit rot，这是一个技术术语，意指随时间推移，软件劣化或者丢失数据的现象。
——译者注

出于一些其他原因，还是有人会认为他们应当现在就编写一些今后才会需要的代码，或者是保留暂时用不到的代码。有些人相信自己可以抗拒变化定律，只要某个功能现在有任何一名用户可能用到，就为此编写代码。这样一来，程序在未来就不需要改变或改进了。但是，这样做是不对的。只要一直有人用，就没有什么系统可以永远保持不变。

还有人认为，现在多做些工作，将来就可以节省时间。有的场合下这种做法行得通，但这不包括写没人用的代码的情况。即便这些代码将来可以派上用场，多半也得要花时间重新设计，所以最终还是浪费了时间。

编写不必要的代码：真实的例子

从前，有位开发人员——就叫他Max吧——以为自己可以不受这条规则的束缚。他的程序里有一项下拉菜单，用户可以从中选取一个值。使用该软件的每家公司都会自己定制这个菜单的选项。有些公司定制的选项可能是颜色的名字，还有些可能是城市的名字，当然还有其他选项。所以，可供选择的项应当存储在每家公司都可以修改的地方。

显然，存储可供选择的值就够了。毕竟，需要做的就是这些。但是Max决定多走一步：既要保存每个选项值，又要记录每个值当前是否“可用”——也就是说，用户现在能否选择这些选项，这个值是否临时禁用了。

不过，Max一直没写任何判断选项是否“可用”的代码。自始至终，无论存储了什么数据，所有的选择都是可用的。虽然Max以为他马上就要写那些判断“可用”的代码了，没准儿就在第二天动手。

几年过去了，判断“可用”数据的代码一直没有出现。相反，数据只是存在那里，没有任何地方用到，却会干扰其他人的认知，产生错误。许多客户和开发人员写信给Max，想询问为什么他们把若干选项设定为“不可用”状态时，没有任何变化。有一名开发人员错误地假设“可用”字段是有用的，甚至写了些代码来使用它，即便系统的其他部分都没有用到。修改后的程序到了客户手里，客户就开始报告奇怪的问题，需要花大力气来追踪。

最后，有些开发人员要动手了，他们告诉自己：今天就要实现禁用选项的功能。但是他们发现，“可用”字段的设计并不能满足自己的需求，所以他得花相当多的时间重新设计才能实现这个功能。

最终的结果是：若干bug，严重的混淆视听和困扰，真正需要这段代码的开发人员还得做大量的工作。这还不算是严重违背规律的行为！倘若严重违背这条规律，会导致更加恶劣的后果，包括项目延期，造成巨大的灾难，整个项目甚至会因此完蛋。

5.2.2 代码难以修改

软件项目的一大杀手就是所谓的“僵化设计”(rigid design)。也就是说，程序员写出来的代码很难修改。僵化设计有两大原因：

(1) 对未来做太多假设

(2) 不仔细设计就编写代码

举例：太多关于未来的假设

一家机构——就叫“老兵医院”吧——希望开发一个程序。他们叫它“医疗保健系统”。在设计该系统之前，医院决定编制文档详细说明整个系统的实现细节，他们花了一年来写这份文档，确定系统中的每一点决策。

然后，开发人员依照文档，花了3年把系统开发出来。在开发时，他们发现文档中的设计有地方自相矛盾，有地方不完整，还有地方很难实现。可是这份文档是医院花了一整年编写的，所以开发人员不能再等一年让医院来修订。于是，开发人员还是尽可能按照文档完成了整套系统。

系统完成之后就进行了首次交付。但是现在，医院的情形已经大不相同了。距离最初的设计已经过去了四年，用户真正使用这套系统时，发现自己需要的已经完全不同了。然而系统是由几十万行代码构成的，其设计也是严格按照文档来构建的，除非再花上几个月甚

至几年，否则系统很难改变。

所以，公司开始为新系统重写设计文档，把之前的过程重新来过。

医院的错误就在于他们试图预测未来。他们假设文档中所作的每一点改动都是对最终用户真正有用的，而且到系统完成的时候还会一直有用。未来真的到来时，却完全与之前预测的不一样，结果几百万美元打了水漂。

更好的办法是每次只确定一个或者少数几个需求，然后立刻让开发人员实现它。在开发过程中，用户可以扮演开发人员的角色，反复进行沟通。上次确定的功能实现并发布之后，就可以继续处理其他的功能。这样，最终得到的系统是设计良好的，完全满足用户需求的。

举例：缺乏设计的编码

某开发人员被要求写一个程序，供人们记录他们完成任务的进度。为了在系统里新建一个任务，用户需要在表单里填写某些信息，比如任务的概要信息、现在的进度。这些信息存储在数据库里。然后，随着时间的推移，大家一点点地记录任务的进展，最终标注自己完成了整个任务。

软件里有个叫Status的字段，用来记录当前任务的进度。这个字段的值有No Work Done（未做），In Progress（进行中），On Hold（暂停），Complete（完成）。如果值为No Work Done，只能改为In Progress。如果值为In Progress，只能改为On Hold或者Complete。如果值为Complete，只可以改回为In Progress。

程序里还有其他10个字段需要用到类似的规则，每个都包含任务某一方面的信息（比如指派给谁、截止日期是哪天）。

为实现这些规则，程序员专门写了一段长长的代码，它没有任何结构，而是独立保存为一个文件。这段代码验证每个字段的正确性，也仅为完成验证功能而设计。比如，每次检查状态是否为

Complete，就在代码里硬编码写上Complete。同样，编写代码时也不考虑重用。遇到相似的字段，开发人员就复制-粘贴代码，然后有针对性地做些小修改。

这段代码当然能用。文件有3000行，但基本与设计绝缘。

几个月过去了，开发人员离开了项目。

之后，一名新开发人员被安排来维护这个项目。他很快发现这段代码很难修改，如果修改其中的一部分，就得对其他很多地方做同样的修改，才能保证程序正常运行。更糟糕的是，这些类似的部分分得很散，既没有注释，也没有逻辑——每次需要修改时，除了阅读整个文件外别无他法。

然后，客户要求添加新的功能。一开始，新来的这名开发人员尽自己努力来完成新功能，他在文件中增加了更多代码。代码行数达到了5000行。

最终，客户要求的某些功能，原有设计方案已经无法实现了。客户希望用电子邮件更新任务的进度，但是原有代码只能处理表单。原有的设计是紧密地与表单绑定的——它无法对接电子邮件。

这时候，竞争对手出现了，他们的任务可以用电子邮件来更新。结果，原有的软件开始流失客户。

这个项目之所以能维持下去，完全是因为有两名开发人员花了一整年重新设计了这个文件，降低了它的修改难度。在重新设计时，他们尽自己努力来满足其他新需求，但大多数时间还是花在重新设计上。[2]

要避免僵化设计，就应当做到：

设计程序时，应当根据你现在确切知道的需求，而不是你认为未来会出现的需求。

注2：Bugzilla中的这个文件叫process_bug.cgi。这个故事做了删略，但是数字（也就是代码的行数，以及用来修复的时间）却是大致准确的。如果你想看关于重新设计整个项目的完整故事，看看这一切是怎么发生的，可以参考这个链接：https://bugzilla.mozilla.org/showdependencytree.cgi?id=367914&hide_resolved=0。

基于当前确知的需求来设计，同时兼顾未来需求的可能性。如果你确切地知道：系统需要完成 X，而且只是 X，那么现在设计时就应该只考虑完成 X。你应该记住，未来系统可能还要提供 X 之外的功能，但是现在，系统只应该完成 X。

如果照这样来设计，另外应该做到的一点是，保持每次改变的幅度都很小。如果要做的改变很小，设计的难度也会很小。

这不是说不要做规划。在软件设计中，一定程度的规划是非常有价值的。但是，即便不做详细的规划，只要你能保持改变的幅度很小，代码也很容易适应不确定的未来，就没有大的风险。

5.2.3 过分追求通用

既然代码将来要修改是一个事实，有些开发人员给出的应对方案就是：做一个足够通用的办法，保证（他们自己相信）可以适应未来任何可能的形势。我们称这种做法为“过度工程”（overengineering）。

按照字典的定义，overengineering 就是 over（意思是“过分了”）加 engineer（意思是“设计和构造”）。根据这种解释，过度工程意思就是，在设计或者构造上花了过多的精力。

等一下，设计和构建过多了？什么是“过多”？设计不是一件好事吗？

是的，大多数工程都应该在设计上花更多的精力，在上一节中“举例：缺乏设计的编码”中已经看到这点。但是总会有人真的掉到过度工程的陷阱里——就如同为了烧毁蚁穴，就去制造一台轨道激光器。轨道激光器是有相当挑战的工程，它耗费不菲，制造时间相当长，维护起来也是一场噩梦。你能想象它出问题的时候应当怎么修复吗？

过度工程还有其他几个问题。

(1) 因为你不能预测未来，所以无论你做得多么通用，其实都不够满足未来要面对的真实需求。

(2) 如果你的代码很通用，那么它通常不能从用户的角度很好地满足规格 / 需求。举例来说，假设你设计的代码把所有输入都当成二进制字节来对待。有时候这代码要处理文本，有时候要处理图片，但是它只知道输入的是字节。某种方面来说，这种设计不错：代码足够简单、自洽、短小。然后你才知道，代码根本无法区分图片和文本——这就是过分通用了。用户传进来一张损坏的图片，得到的错误却是“传入的字节错误”。本来它应该报告“图片已经损坏”，但是你的代码太过通用，无法给出这种提示（太过通用的代码很多时候无法满足具体的需求，这只是一个例子）。

(3) 太过通用就必须写很多不需要的代码，这样又回到了第一条规律。

总的来说，如果你的设计让事情更复杂而不是变简单，就是在做过度工程。如果你只需要清理蚁穴，动用轨道激光就会把事情大大搞复杂，其实只用一点蚂蚁药就可以解决问题（假设它有效）。

在追求通用时，应当选择正确的事情，选择正确的方法，这是成功的软件设计的基础。然而，太过通用，会带来说不完的复杂和混乱，也会大大抬高维护成本。避免此误区的办法，和避免僵化设计的一样：

仅仅根据目前确知的需求来考虑通用。

举例：太过通用

某程序的某个部分的功能是，用户填写一张表单，程序就发送几百封邮件。这个部分的程序运行起来很慢。用户提交表单之后，程序会在这个环节停留很长时间，等待发送完所有的邮件。

为了提高这个环节的速度，开发人员决定不要立即发送所有的邮件，而是改为在后台发送。用户提交了表单之后，程序会使用之前存在的“Email Sender”代码来发送邮件。

负责这个任务的开发人员认为有些公司并不会使用Email Sender。他写了几百行代码，这样用户就可以把其他系统以插件的方式塞进来，在后台执行这个任务。从没有客户提过这个需求，开发人员只

是猜测未来会有人要求这种灵活性。

最终，首席架构师叫停了这种做法。他去掉了所有与“插件”有关的代码，因为根本没有证据表明有人会那么用。也就是说，没有任何证据表明目前代码需要做到那么通用。这些代码删掉之后，修改变得容易多了。

这一改变已经过去了四年，期间没有任何客户要求给系统做插件。也就是说，系统其实根本没有必要做得那么通用。

5.3 渐进式开发及设计

有个办法可从根本上避免这三大误区，这就是“渐进式开发和设计”。它要求按照特定顺序，一点一点地设计和构建系统。

这个办法很容易举例来说明。假如需要开发可以计算加减乘除的计算器程序。

(1) 设计一个只能进行加法运算的系统。
(2) 实现它。
(3) 修改现有设计，很容易就可以支持减法运算。
(4) 实现减法运算。现在，系统只能进行加、减运算。
(5) 再次修改现有系统的设计，很容易就可以支持乘法运算。
(6) 实现乘法运算功能。现在的系统可以支持加、减、乘运算了。
(7) 再次修改系统的设计，添加除法运算（现在做这一步应该不需要花多少功夫了，因为我们早就改进了设计，添加过减法、乘法）。
(8) 实现除法功能。这时候得到的就是一开始期望构建的完整系统，而且拥有满足需求的良好设计。

相比开始就建立完整的系统，一次性构建出来，这种开发方法需要时间更少，也不用考虑过多。如果你习惯其他的开发方法，头一次实践可能并不那么容易，但是经过锻炼，用起来就会变容易。

这个方法的精妙之处就在于，它是根据实现的顺序来决策的。总的来

说，在其中的每个阶段，下一步都只做最容易的事情。首先选择加法，因为这是最简单的运算；其次选择减法，因为从逻辑上说，它与加法只有很小的差异。第二步也可以选择乘法，因为乘法无非是把加法执行很多次而已。这时唯一不应当选择的就是除法，因为从加法到除法的步子距离太远了，步子太大了。而且，最后一步从乘法到除法是非常简单的，所以这是一个好的选择。

有些时候，你甚至需要把某个单独的功能拆分为一系列小的、简单的逻辑步骤，然后才可以很方便地实现。

显然，这里混合了两种做法：一种叫做“渐进开发”，另一种叫做“渐进设计”。渐进开发是一种通过小步骤构建整个系统的办法。在上面的步骤里，“实现”开头的每一步都是渐进开发过程的一部分。渐进设计是一种类似的方法，它也通过一系列小步骤用来创建和改进系统的设计。在上面的步骤里，以“修改”或“设计”开头的每一步，都是渐进设计过程的一部分。

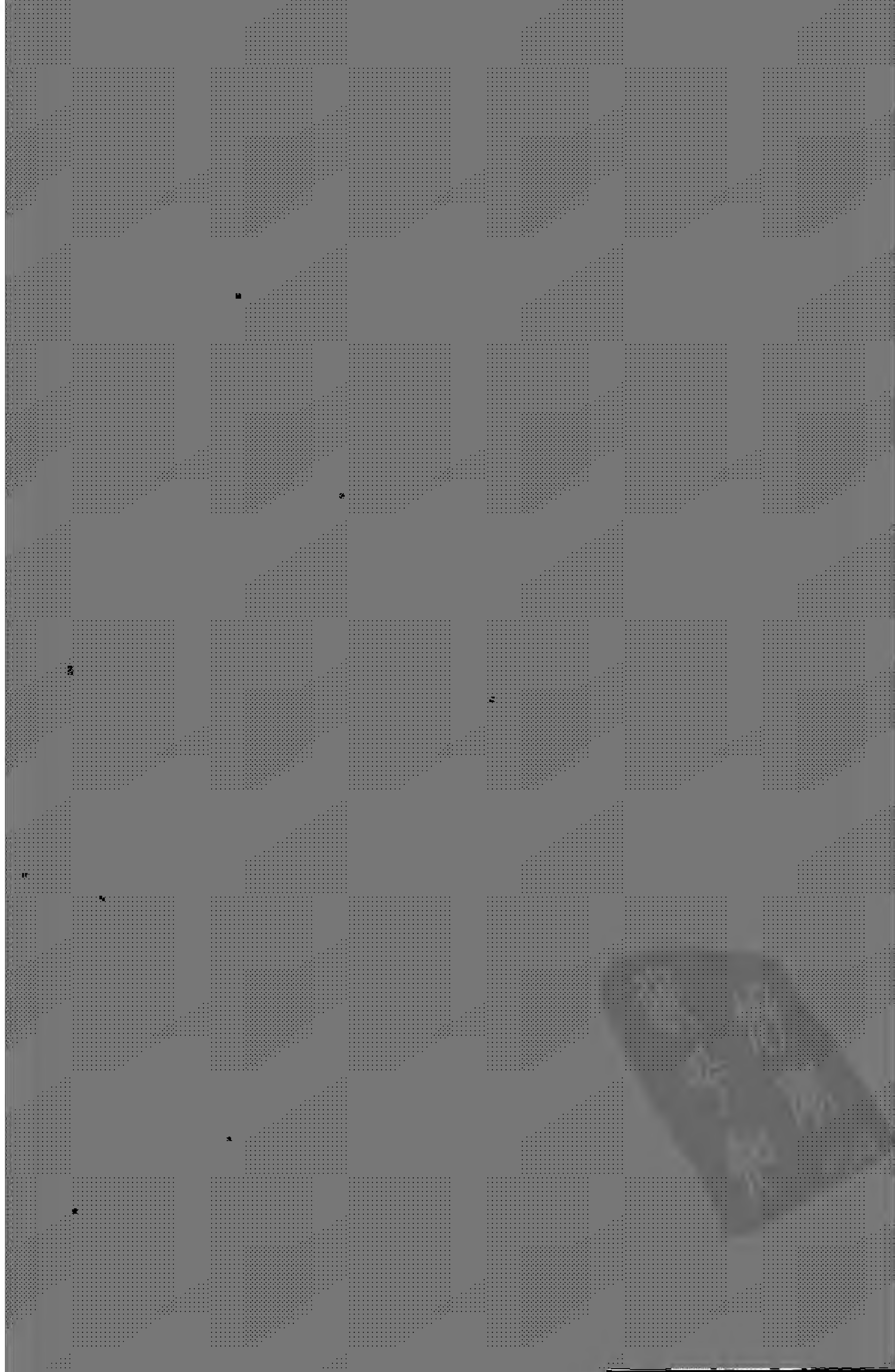
渐进开发和设计并不是唯一有效的软件开发方法，但是它无疑可以避免之前列出的三大误区。



第6章

缺陷与设计





很不幸，没有哪个程序员可以不犯错误。不错的程序员大概每写 100 行代码就会犯一个错误；最优秀的程序员，在状态最好的时候，每写 1000 行代码也会犯一个错误。

也就是说，无论程序员的水平是高还是低，有一条是不变的：写的代码越多，引入的缺陷就越多。这样，就可以得出“缺陷概率定律”：

> **在程序中新增缺陷的可能性与代码修改量成正比。**

这一规则之所以重要，是因为错误妨碍了我们帮助他人的目标，故而应当避免。而且，修复缺陷也是维护工作的一种。所以，新增缺陷还会抬高维护成本。

既然存在这条规则，又无法预测未来，我们很快就会发现，相比大的变化，小的变化维护成本更低。小的变化 = 更少的缺陷 = 更少的维护。

有时候，该规则也会被非正式地表述为：“如果不新加代码，也不修改代码，就不会产生新缺陷。”

这条规则的有趣之处在于，它似乎与“变化定律”相矛盾——软件必须要变化，但是变化又会引入缺陷。这种矛盾确实存在，如何在两者间取得平衡，取决于软件设计师的聪明才智。实际上，这种矛盾恰恰说明了为什么需要设计，而且告诉我们，理想的设计是怎样的：

> **最好的设计，就是能适应外界尽可能多的变化，而软件自身的变化要尽可能少。**

这个说法，简练融合了如今关于优秀的软件设计的各项知识。

6.1 如果这不是问题……

没错，如果不添加代码，也不修改代码，就不会引入新的缺陷，这是软件设计中的一条主要规律。不过，还有下面这条非常重要的规律与之相关，许多软件设计师都听过某种形式的表述，虽然他们有时候会忘记：

永远不要“修正”任何东西，除非它真的有问题，而且有证据表明问题确实存在。

在动手修正问题之前，获得证据是很重要的。否则，你的辛苦努力很可能解决不了任何人的问题，或者你很可能会“修正”根本没有问题的代码。

如果没有获得证据就动手修正问题，很可能只会添乱。修改现有系统，可能会造成新的错误。而且，这么做还会浪费时间，增加程序的复杂性，却没有任何道理。

那么，什么是“证据”呢？假设有五名用户反馈：一点击红色按钮，程序就崩溃了。好，这就是足够充分的证据了！否则，你可能需要自己去点击红色按钮，看看程序是否崩溃。

但是，如果只有一个用户报告错误，并不能说明这就是问题。有时候用户不知道你的程序已经提供了某些功能，所以希望你去重复实现它们。举例来说，你的程序可以按照字母顺序来排序一系列单词，而用户希望你添加一个功能，能按字母顺序排序若干字母。这个功能程序里早就有了，而且其实比用户想要的还要强大——通常情况都是这样，只是用户没弄清楚到底要干什么。在这个例子里，因为用户没有发现单词排序的功能，他可能认为没有实现字母排序是个问题。用户还可能给出“证据”，说明自己无法排序一系列字母。其实，问题仅仅在于，他没有意识到自己应该使用单词排序的功能。



如果你收到很多这样的请求，就说明用户很难在你的程序里找到自己想要的功能。这才是真正要改进的地方。

有时候用户会报告某个 bug，但程序其实是完全按照预期来运行的。果真如此，就应当少数服从多数。如果相当多的用户认为某个行为是 bug，它就是 bug；如果只是少数用户（比如一两个）认为它是 bug，那么它就不算 bug。

在这类问题上，最有名的错误就是所谓的“提前优化”。也就是说，

有些开发人员想让速度尽可能快，所以，他们还没弄清楚速度到底慢不慢，就花时间来优化程序。这就好像做慈善事业时，一边把食物送给富人，一边说“我们只是希望帮助他人”。这不合逻辑，对吧？因为这样是在解决根本不存在的问题。

在你的程序中，真正需要关注速度的部分，应该局限于你可以证明的、真正让用户体会到有性能问题的那些部分。对程序的其他部分，最主要关心的还是灵活和简洁，而不是速度。

要违背这条规律有成千上万种办法，不过遵守它的办法很简单：在动手解决之前，真正拿到证据，证明问题确实存在。

6.2 避免重复

在软件设计中，这或许是最著名的条例。其他资料也曾提过该条例，鉴于它的重要性，我们在这里重申：

> **理想情况下，任何系统里的任何信息，都应当只存在一次。**

假设有个 Password 字段，在你的程序里会出现在 100 个用户界面上。如果你希望把它改为 Passcode 该怎么办？如果字段名统一存储在程序某个地方，就只需要修改一行代码。但如果每次都是硬编码写上 Password，就得修改 100 次。

这个道理对代码段也同样适用。我们不应该复制粘贴代码段；相反，应该使用各种编程技巧来处理，让各处的代码可以“使用”（use）、“调用”（call）、“包含”（include）已有的其他代码。

遵守这条规则的一个强有力的理由，就是缺陷概率定律。如果新增功能时可以重用代码，就不需要写太多代码，引入错误的可能性也就随之减少了。

这同样有益于设计的灵活性。如果我们需要更改程序的运行结构，就可以修改某一部分，而不是查遍整个程序，在各处修修补补。

众多优秀设计都基于这一规律。也就是说，你能更聪明地让代码“使用”其他代码，把信息集中运用好，那么设计也就更好。在这一领域，你同样可以真正用自己的聪明才智为编程创造价值。



第7章

简洁



现在我们知道了，如果软件一直不变化，就可以彻底避免出现新错误。然而，软件必定是要变化的，尤其是需要增加新功能时，变化是不能避免的，所以“一直不变化”并不是彻底杜绝错误的办法。

第 6 章讲过，如果希望避免代码出现新错误，可行的办法之一就是把变化的规模限定在小范围内。不过，如果既要做很多修改，又希望这些变化不要引入错误，还可以用上另一条法则。它不仅仅用来消除错误，还可以保持程序的可维护性，降低新增功能的难度，让代码更容易理解。这就是简洁定律（Law of Simplicity）：

软件任何一部分的维护难度，反比于该部分的简洁程度。

换句话说，某一部分的代码越简洁，未来进行变化的难度就越低。完全消除维护的难度是不可能的，但这正是我们要争取实现的目标：如果要进行彻底的变化，或者新增大量代码，不应该遇到多少困难。

你可能注意到了，这条法则并不关心整个系统的简洁性，只是谈到了各个部分的简洁性。这是为什么呢？

原因在于，一般的计算机程序已经足够复杂了，没有人可以一次性全面理解它，大家都只能分部份逐步了解。虽然程序里大都有些庞大繁杂的结构，但这不要紧；要紧的是，在我们阅读代码时，应该可以理解这些庞大繁杂的结构。这些部分越简洁，就越容易被普通人理解。这一点非常重要，尤其是你把自己的代码转交给其他人，或者脱离自己的代码几个月再重新上手时，更是如此。

用建筑结构来类比

假设要搭建30英尺高的钢铁建筑。你可以用很多短撑杆，也就是小零件来搭建；也可以先生产3个巨大而复杂的构件，再把它们组合起来。

如果用短撑杆来搭建，零件是很容易生产也容易买到的。假如某个零件坏掉了，也很容易找到备件来替换。整个建筑过程是简单的，维护也简单。

如果采用3个大构件，就需要花费大量精力来细心定制。因为每个构件都很大，因此很难找到并修正它的缺陷。而且，如果建筑完工之后发现构件上有若干问题，你也不能换掉它——抽掉任何一个构件，整个结构就会倒塌。所以你只能依赖丑陋的补丁，并且祈祷整个结构不会出问题。

软件的情况与之类似。如果你的代码写得简洁、自洽，那么修正问题、维护系统就很容易。如果你设计的是庞大繁杂的模块，那么每一部分都需要花费大量精力，也很难做到足够精致。这样的系统就很难维护，而且必须经常打补丁才可以正常运行。

那么，为什么会有人编程时要采用庞大繁杂的模块，而不采用小巧简洁的模块呢？在软件第一次编写时，采用大模块表面上可以节省下相当多的时间。如果使用众多小模块，就必须花很多时间来组合。如果采用大模块则不会这样，零件很少，而且很容易对接。

不过，由大模块构成的系统，质量要差得多，而且，将来你得花很多时间去修正错误，结果就是维护的难度越来越高。相反，简单系统的维护难度会越来越低。长期来看，能保证效率的恰恰是简单的系统，而不是复杂的系统。

那么，在真正编程的时候，要如何落实这条法则呢？本书的其余章节很大篇幅都在讨论这个问题。不过总的来说，核心思想就是要让代码中各个部分都尽可能简洁，并且尽力一直保持这种简洁性。

落实这条法则的一个好办法，就是第 5 章讲解的渐进式开发和设计方法。因为每次添加功能之前都有个“重新设计”的过程，所以系统能持续简化。即便不用这种方法，你也可以在增添新功能之前，花点时间去化简任何让你或你的同事觉得不够简洁的代码。

无论如何，你的代码一般总是归你负责，化简也归你负责——你不能奢望最初的设计永远都是最恰当的。在新的形势和需求面前，你必须不断重新设计系统的各个部分。

没错，这个任务相当困难。不可能任何时候都有简单的工具来写程序——编程语言是复杂的，计算机本身也是复杂的。不过，我们还是应当尽力追求简洁。

7.1 简洁与软件设计方程式

虽然你可能已经意识到了，这里还是要说，简洁定律告诉我们：目前可行的、能够降低软件设计方程式中维护成本的最重要的事情，就是把代码变简洁。我们不必预测未来，完全可以只审视自己的代码，如果它足够复杂，就立刻动手简化它。这就是随时间推移降低维护成本的办法——持续不断地让代码变得更简洁。

进行这种简化有一定的工作量，但是总的来看，对简洁系统做修改总是比复杂系统要容易很多，所以，现在花一点时间去追求简洁，将来就可以节省大量的时间。

随着系统维护成本的降低，各种想做的修改的可行性也会增加。（如果你希望重温细节，请翻到第 4 章看看软件设计的方程式。）化简代码可以降低维护成本，也就增加了各种可能修改的可行性。

7.2 简洁是相对的

没错，我们希望把代码变简洁。不过如何定义“简洁”，却取决于你的目标受众。你觉得某个东西简洁，你的同事却可能不这样认为。另外，你可能觉得自己创造的某样东西很“简洁”，因为你了解得非常透彻。但是在完全没接触过的人眼里，它可能无比复杂。

如果你希望知道第一次看你代码的人有什么看法，不妨找些从没接触过的代码来看看。你需要理解的不只是那一行行的代码，而是整个程序在干什么；而且你还必须弄清楚，要做修改该如何动手。其他人看你的代码的时候，就是这种感觉。你可能发现了，如果阅读别人写的程序，就算不很复杂的代码也会让人相当郁闷。

所以，一个不错的做法就是在自己的代码注释中加上类似“头一次看

这段代码？那么……”这样的注释，在其中给出一些概略的解释，帮助其他人理解。写注释时应当考虑到，读者完全不了解这段程序，因为如果是你初次接触某些东西，你恐怕也对它一无所知。

无数的软件项目在这方面表现得一团糟。翻开写给开发者的文档，你只能看到大堆的链接，而没有任何头绪。如果开发人员已经在这个项目上工作了很长时间，事情就非常简单，因为这张有大量链接的索引表，可以帮他们迅速找到要看的内容。但是，这种文档对新手来说非常复杂。而对工作了很长时间的开发人员来说，如果在文档中新增一页，在上面用简单醒目的按钮取代原本熟悉的链接，只会把事情搞复杂，因为他要做的只是在文档中迅速找到相应的资料。

比起繁杂的文档，更糟糕的情况就是没有文档，开发软件的人认为其他人可以自己找出问题，或者“早就知道”程序是怎样运行的。对开发人员来说，自己写的程序的原理是显而易见的，但对其他人来说，这是一个完全陌生的世界。

应用场合（上下文）也是非常重要的。在程序代码中，先进技术如果使用得当，通常会让代码简洁。但是，如果程序的复杂内部结构只有通过某个 Web 页才能看到，其他方式都不可行，这就算不上简洁了，即便对开发人员来说，也不算简洁。

有时候，某个应用场合中看来复杂的东西，换个应用场合就会变简单。在马路边的广告牌上标注大量解释文字是非常复杂的——过路的司机根本来不及去看那么多文字，所以这么做很愚蠢。但是在程序使用手册里，给出大量的解释性文字就比一句话的概要描述简洁得多。所以，本书的每章并不会一句话就完事。干巴巴地写几句话没有解释，并不算简洁。

考虑到不同的视角和应用场合，追求简洁是否极其困难呢？不，绝对不是。我们做的每一件事都会有目标受众，每个应用场合通常都存在诸多限制。所以，追求简洁是绝对可行的。重要的是要在设计软件时考虑到上述这些因素，这样其他人在真正使用时就会感到软件是非常简单的。

编辑器之战

完成某项任务的最好用的工具是什么，在软件开发的世界里，这是个争论不休的问题。各人偏爱的文本编辑器不同，编程语言不同，操作系统也不同。在软件开发中最著名的“战争”，大概就是两款文本编辑器之间的战争，这两款编辑器就是vi和Emacs。每一派的用户都宣称，自己喜欢的编辑器从根本上讲要优于另一款编辑器。

其实在编写代码这件事上，基本没什么工具能从根本上强过另一款。实际的情况是，特定的用户觉得用某些工具更容易解决手头的具体问题。Emacs用户认为Emacs是最简洁的编程工具，而vi的用户认为vi是最简洁的。从某种程度上讲，之所以会有这样的局面，是因为人的喜好是有巨大差别的，大家喜欢的工作方式不同，思维方式也不同，大家的偏好必然有差异，这并没有对错可言。但是更深入一点来看，工具的简洁程度是与每个人的使用熟练程度相关的。任何人，只要使用某款工具足够长的时间，都会更熟悉它，从个人角度出发，会觉得它比其他工具更简洁。如果要求一款新工具表现得同样简单，那么这款工具必须简洁到极致。显然程序员用的文本编辑器做不到这一点。

不写程序的人可能觉得两款编辑器都太复杂了，已经无法理解，这个例子再次说明，简洁是相对的。



如果工具不适用于手头的某个任务，或者在设计中做了错误的选择（参见 8.2 节），此工具就可能带来问题。如果忽略这些问题，工具的相对简洁性就在于，在某个具体的情况下，程序员可以自行选择用起来最舒服的工具。

7.3 简洁到什么程度？

一旦开始做某个项目，就会遇到关于简洁的问题。我们要简洁到什么程度？需要把事情简化到什么程度？它已经足够简洁了吗？

没错，简洁是相对的。不过即便如此，还是存在更简洁和更复杂的差

别。从用户的角度出发，你的产品可能很难用，也可能很容易使用，还可能介于两者之间。同样道理，在其他程序员看来，你的程序阅读起来也可能比较困难，或者比较简单。

那么，要简洁到什么程度呢？

要听真话吗？

你真的想要成功？

简单到傻子也能懂！

关于简洁程度，有一点相当有意思：在大多数场合，任何普通人可以用到的，天才也可以用。所以，软件的可能用户的类型，或许比设想的要多得多。

但是大家通常不理解，怎么做才算真正简单到傻子也能懂。举个例子：大商场里都有一张示意图标识方位。在最清楚的示意图上会有一个大红点，并在你面前用醒目字体标识“你在这里”。糟糕一点的示意图中间会有一个不易发现的小黄三角形，在示意图侧边用文字说明“小黄三角形表示‘你在这里’”。这只会让看示意图的人更困惑，你可能需要花五六分钟才能找到想去的地方。

对设计示意图的人来说，出现这种情况并不难理解。既然花了大量的时间来设计，那么对他来说，这张示意图显然足够重要，他乐意花几分钟来查看和学习，找到自己想要的信息。但是对其他人想的只是看看这张图，所以根本不关心花了多少时间来设计。用户只希望尽可能简单，这样才能迅速上手，才能真正用起来。

许多程序员在这方面做得尤其差劲。他们以为别人都愿意花很多时间来学习自己写的代码，毕竟这是自己花很多时间写出来的。这些程序员很重视自己的代码，所以对其他人也应当同样重视。

没错，程序员通常都是非常聪明的人。但是“噢，其他程序员会理解我写的所有代码，没必要简化或者注释”的看法仍然是不对的。这个问题与智商无关，而与背景知识有关。第一次接触你代码的程序员完

全没有任何背景知识，他必须学习。学习的难度越低，找出问题的速度也就越快，使用起来也越容易。

降低代码学习难度的方法有很多：简洁的注释，简单的设计，循序渐进的引导，等等。

不过，如果你的代码没有做到傻子都能看懂，其他人学起来就会遇到困难。他们会误解，会制造 bug，会把事情搞得一团糟。等这一切发生的时候，他们会找谁？对，就是你。这时候你就得花时间回答他们的各种问题。（嗯，听起来挺讽刺的，对吧？）

我们谁也不希望被当成傻子来训导或对待。所以有时候我们要搞出些稍微复杂点的东西，这样才显得比用户或其他程序员更聪明。我们夸夸其谈，故意把软件做复杂点，让别人膜拜我们的智商，让他们因为自己搞不懂而感到愚蠢。他们可能觉得自己永远也不会有我们那么聪明，这确实让我们有种成就感。可是说真的，这样做能帮助他们吗？

相反，如果你的产品或代码简单到傻子都懂，大家就都可以理解它。这样，其他人会感觉自己是聪明的，也可以完成自己希望完成的任务，同时你并没有增加多少负担。其实，如果你把事情做得简单而不是复杂，大家甚至会更羡慕你。

当然，你的家人没必要能读懂你的代码。所以，简单仍然是相对的，你的代码的目标受众不是家人，而是其他程序员。但对这些程序员来说，你的代码应该尽可能简洁，容易理解。编程过程中完全可以使用所需的各种先进技术来达到这种简洁，但是代码本身必须是简洁的。

面对“要做到多简洁”这种问题时，你可能需要同时问自己：“我究竟是要让用户理解，感到快乐，还是让他们困惑，感到沮丧？”如果选择前者，确保成功的复杂度就只能是：简单到傻子都能懂。

7.4 保持一致

要做到简单，保持一致是很重要的工作。如果你在一个地方采用了某种规则，就应当在其他每个地方都遵守这种规则。

如果某个变量命名为 somethingLikeThis，那么所有的变量都应该遵守这种规则（其他变量名应该是 otherVariable、anhtoerNameLikeThat 等）。如果变量命名为 named_like_this，那么所有的变量都应该用小写，并且用下划线连接这些单词。

如果代码不能保持一致，程序员理解和阅读起来都要更加困难。

还可以举自然语言的例子说明这个道理，看下面两句话：

- This is a normal sentence with normal words that everybody can understand.
- tHisisanOrmalseNtencewitHnorMalwordsthAtevErybOdycAnunderStaNd.

两句话的意思相同，但是第一句更容易阅读，因为它符合大家的书写习惯。当然，第二句也不是看不懂，但是你真的希望整本书都写成这个样子吗？同样的道理，如果某个程序毫无一致性可言，你愿意去看吗？

在编程时，有时候你做什么并不重要，只要一直保持相同的方式去做即可。理论上说，你当然可以把代码写得无比复杂，但是也要保持复杂的一致性，这样其他人才可以阅读。（当然，更好的办法是保持一致并且做到简单，但是如果达不到极其简单，至少要保持一致。）

在许多情况下，完全保持一致可以让编程更简单。如果程序中每个对象都有一个 name 字段，你就可以写一小段简单代码来处理整个程序中所有对象的 name 字段。但是如果在对象 A 中它叫做 a_name，在对象 B 中它叫做 name_of_mine，你就得为对象 A 和对象 B 分别做特殊处理。

同样，程序的内部行为应当保持一致。如果你已经熟悉了系统的某个部分，就应当可以迅速熟悉其他部分的代码，因为每部分的风格都是类似的。如果要使用 A 部分，程序员需要调用三个函数，再写一些代码；那么为了使用 B 部分，应当调用和那三个函数类似的几个函数，再写一些代码。如果在 A 部分中，有个叫 dump 的程序可以打印出所有内部变量，那么 B 部分中的 dump 函数应当完成同样的功能。千万别让程序员每次遇到新的部分都重新学习。

真实世界里或许不存在这样的一致性，但是程序的世界由你负责，所以必须保持程序的简单和一致。

在真实世界中也有些一致性的例子。在亚洲大部分国家，人们用筷子吃饭。在欧美国家则使用刀叉。虽然有两类餐具，但是总的来看，在某个地区还是相当一致的。假设每次你去别人家，都必须学会使用新的餐具，Bob 家用的是剪刀，Mary 家用的是硬纸板，这样吃饭就成了大问题，不是吗？

编程也是这样——缺乏一致性，只会一团糟。有了一致性，世界就很简单。即便你做不到那么简单，至少也要做到：一旦你理解了某种复杂性，就不必再进行重复劳动。

7.5 可读性

软件开发领域反复强调一点：代码被阅读的次数远多于编写和修改的次数。所以，保证代码容易阅读很重要。

> **代码可读性主要取决于字母和符号之间的空白排布。**

如果世界是一团漆黑，就分不出任何物体，因为万物都是黑色的一团。同样道理，如果整个文件的代码乱成一团，没有格式稳定、符合逻辑的空白，就很难拆分各个部分。要把各部分拆分开来，就必须留出空白。

空白太多是不必要的，因为这样很难发现事物之间的联系。空白太少也不必要，因为这样很难分解。代码到底应该留出怎样的空白，没有什么硬性的、直接的规则，唯一的规则是，代码之间留出的空白应当保持一致规范，空白应当能有助于读者理解代码的结构。

示例：空白排布

下面的代码很难看懂，因为空白太少了，几乎没有提供关于代码结构的信息。

```
x=1+2;y=3+4;z=x+y;if(z>y+x){print"error";}
```

同样的代码，这里的空白又太多了，读者很难理解代码的结构。

```
x              =            1+        2;
y = 3               +4;

  z = x     +      y;
if (z  >     y+x)
 {         print "error" ;
        }
```

写成上面这样，甚至比没有留出空白的代码还要难看懂。

下面这段代码的空白设置得比较合适：

```
x = 1 + 2;
y = 3 + 4;
z = x + y;
if (z > y + x) {
    print "error";
}
```

这样读起来就容易多了，也有助于大家理解程序员写这段代码的目的。首先给3个变量赋值，然后根据某个条件，显示某个错误。程序员通过安排合适的空白，把代码的逻辑结构清楚地向读者表达出来。

如果代码很容易阅读，也就容易修正。在前面的例子中，如果空白留得合适，我们可以很容易地发现，z永远不可能比`y + x`大，因为z永远等于`y + x`。所以，`if (z > y + x)` 开头的这段代码其实是没有用的，应当删掉。

一般来说，如果某段代码有很多bug，又难以阅读，那么首先要做的是让它更容易阅读。然后，bug在哪里才能看得更清楚。

7.5.1 命名

可读性的另一部分重要内容是为变量、函数、类等选择合适的名字，理想的命名应该这样：

> **名字应当足够长，能够完整表达其意义或描述其功能，但不能太长，以免影响阅读。**

同时，还应当考虑函数、变量等的使用频率。在代码中使用这些名字，是否会导致代码过长而影响阅读？比如，如果有个函数只会在某一行由你自己调用一次（而且这一行没有其他代码），它的名字可以很长。不过，如果某个函数会经常在复杂的表达式中用到，大概就该取个短一些的名字（但还是要足够长，能够描述它的功能）。

示例：命名

这段代码的命名就很糟糕：

```
q = s(j, f, m);
p(q);
```

这些名字没有说明变量的用途和函数的功能。

这段代码与之相同，但命名很不错：

```
quarterly_total = sum(january, february, march);
print(quarterly_total);
```

还是同样的代码，但名字太长，难以阅读：

```
quarterly_total_for_company_in_2011_as_of_today =
add_all_of_these_together_and_return_the_result(january_
total_amount,
february_total_amount, march_total_amount);
send_to_screen_and_dont_wait_for_user_to_
respond(quarterly_total_for_company_in_2011_as_of_today);
```

这些名字占据了太多空间，很难看明白。所以从某个角度来看，问题又回到了字母和符号之间应该怎样留出空白。

7.5.2 注释

为保证代码的可读性，好的的注释也很重要。但是，代码的意图通常不应该用注释来说明，直接阅读代码就应当能够理解。如果发现意图不够明显，那么就说明这段代码还可以变得更简单。如果你的代码实在不能更简单，才应该写注释来说明。

注释的真实目的，是在理由不够清晰明显时加以解释。如果不解释，其他程序员在修改这段代码时可能会很困惑；如果不明白这些理由，他们可能会删改其中重要的部分。

有些程序员相信，可读性是追求代码简洁性的全部和终极目标，如果写的代码很容易阅读，你就已经做完了设计师要做的一切。但事实并非如此，你的代码可能很容易阅读，但系统仍然非常复杂。不过，保证代码的可读性是非常重要的，而且它往往是通往优秀设计之路上应当走出的第一步。

7.6 简洁离不开设计

不幸，没有人天生就会构建简洁的系统。如果设计师不倾注精力，系统就会逐渐变成杂乱庞大的怪物。

如果项目没有好的设计，而且持续膨胀，最终的局面会复杂得让你头疼。有些人可能想象不出来这种场景，就像有些人预计不到午饭后会发生什么，或者有些人缺乏足够的经验来理解最终的复杂局面。还有些企业的文化是“噢，我们只不过缺几项新功能；做事是应当规范点，然而……但是……可惜……”不过，总有一天你的项目会失败。无论你可以找到多少理由，失败是避免不了的。

反过来，如果你的设计非常好，一般却难得听到多少称赞。设计中的缺陷是大家都看得到的，但逐步演变为良好设计的改进过程，却是不熟悉代码的人看不到的。于是，设计师就成了一个费力不讨好的工作。解决重大缺陷为你赢得很多赞誉，但是避免缺陷发生……嗯，没有人会注意到。

所以，让这本书来表扬你吧。关于设计，你有没有多加思考？有？干的不错！你的用户和同事会感觉到好处——运行流畅的软件，按时的发布，条理分明、容易理解的代码。你对自己的工作有足够的信心，完工之后充满了成就感。其他程序员知道要花多少精力才让一切有条理吗？可能不知道。但是，这不要紧。除去你周围人的称赞，还有其他的奖励。

只有在个别情况下，你的工作才会赢得其他人的称赞。但是别绝望，最后总会有人注意到的。在这之前，你应当享受有效、正确的设计所带来的各种积极结果。



如果把本书中提到的设计原则应用到手头的项目中，经验不多的程序员或同事可能要花很长的时间才能理解，他们为什么同样需要追求优秀的设计。请他们来读这本书可能会帮上点忙，如果他们不能或不愿意，应当持续引导他们（如果实在不得已，就强迫他们）追求优秀的设计，他们会在（最多）几年后认识到，优秀的设计能带来怎样的好处。



第8章

复杂性





身为职业程序员，你很可能听说过（或者经历过）常见的程序开发噩梦："五年前项目启动时，这项技术还是很先进的，可是现在过时了。因为技术已经被淘汰，事情越变越复杂，所以项目完成的希望越来越渺茫。但是如果推倒重来，可能还要等上五年。"

还有一个情况也很常见："我们的开发速度不够快，跟不上现在用户的需求。""我们正在开发，X 公司就以更快的速度完成了比我们更好的产品。"

我们现在知道，这些问题的根源都在**复杂性**。开始的时候项目是简单的，只要一个月就能完成；然后复杂性增加了，于是需要三个月时间；再然后，每个部分都更加复杂，所以项目需要九个月才能完成。

复杂性是会叠加的，而且不是简单的线性叠加。也就是说，下述假设是不成立的：之前有 10 项功能要开发，因此再加 1 项只会增添 10% 的工作量。因为新增的功能需要与已有的 10 项功能相协调，所以如果开发新功能需要 10 小时，可能还要花 10 小时才能保证已有的 10 项功能与新功能正常交互。原有功能越多，新增功能的成本就越高。优秀的设计可以尽量避免此类问题，但是每项新功能仍然会有单独的成本。

有些项目从一启动就设定了繁多的需求，所以永远无法发布第一版。如果遇到这种情况，就应当删减功能。初次发布不应当设定过高的目标，而应当先让程序跑起来，再持续改进。

除了新增功能，其他一些做法也会增加复杂性，以下列出了最常见的做法。

1. 扩展软件的用途

一般情况下，应当绝对禁止这样做。市场部可能巴望着某款软件既能够计算个税，又可以充当菜谱，这样的需求，你必须尽全力抵制。软件应当坚守已经确定的用途，只要妥善完成这些目标，你就会获得成功（前提是该软件能帮到用户，切实满足其需求）。

2. 新增程序员

没错，往团队里增加新人并不会让事情变简单，相反会更复杂。Fred Brooks 的名作《人月神话》说的就是这个道理。如果已经有了 10 个开发人员，再增加 1 个人，就意味着需要为他设定合适的岗位，花时间让之前的 10 个人适应新人，花时间让新人学会与那 10 个人沟通，如此等等。相比众多平庸的开发人员，少量精干的开发人员更容易获得成功。

3. 做无谓的改变

每做一点改变，都会增加复杂性。无论是需求变化、设计变化，或是只修改某段代码，都有可能增加新的 bug，另外别忘了算上决定如何变化所需的时间，实现它所需的时间，验证它是否影响到原有系统所需的时间，记录它的时间，测试它的时间。每做一点新变化，整体复杂性就会增加一点，所以变化越多，每个变化要花的时间就越长。做出某些变化是重要的，但是应当谨慎决策，而不是一拍脑瓜就定了。

4. 困于糟糕的技术

一般来说，“困于糟糕的技术”指的是你之前决定了采用某种技术，因为极度依赖它，长期无法摆脱。这里说的“糟糕”，意思是你深陷其中（未来无法简单地切换到其他技术），不能灵活地适应未来的需求，或是达不到设计简洁软件所需的质量标准。

5. 理解错误

程序员不理解自己的工作，就容易设计出复杂的系统。这可能是恶性循环：理解错误导致复杂性，复杂性又进一步加剧理解错误，如此往复。提升设计水平的最主要办法是，确保自己完全理解所用的系统和工具。你对它们的理解越到位，对软件开发的一般规律了解越多，你的设计就越简洁。

6. 糟糕的设计或不做设计

一般来说，它指的是“没有为变化做计划”。万物都是会变化的，项目

增长时，设计仍然要维持简单。你必须一开始就做好设计，而且在系统膨胀时不断进行优秀的设计；否则，复杂性就会迅速增长，因为如果设计得不好，每项功能都会让代码加倍复杂，而不是只复杂一点点。

7. 重新发明轮子

如果有相当不错的现成协议，还要自己发明协议，那么仅仅为了把软件跑起来，这些协议也会花去你大量的时间。决不要什么都靠自力更生，去自己开发什么 Web 服务器、协议或者重要的类库，除非它们是你的最终产品。只有在满足以下任何一个条件的前提下，重新发明轮子才有价值：

(1) 你需要的东西不存在；
(2) 现有的各种“轮子”都很糟糕，会把你困住；
(3) 现有的“轮子”根本无法满足你的需求；
(4) 现有的“轮子”缺乏良好的维护，而你也不能接过维护的任务（比如，你没有源代码）。

这些因素都会逐渐影响你的项目，但不会在短时间凸显。它们大多数会造成长期的负面影响，甚至一两年内都觉察不出来，所以如果有人指出这些问题，通常大家也不觉得有什么坏处。即便你走上这条路，重新发明的轮子可能看来也没有问题。但是随着时间的推移，尤其是这些轮子堆积起来后，复杂性会越发明显，不断累加，最终你就成了那个广为人知的故事的受害者，产品永远也发布不了。

8.1 复杂性与软件的用途

你正开发的任何系统，其基本用途应当相当简单。这样开发出来的系统，既满足实际需求，整体来说也是简单的。但是，如果你给系统添加新功能去满足其他目标，事情就立刻变复杂了。举例来说，文字处理软件的基本功能就是帮助用户写作文档。如果突然要求它能够阅读邮件，最终就会得到的非常荒唐复杂的玩意。你能设想它的用户界面吗？各个按钮都要放在哪里？我们都知道，这不是文字处理软件本来

的用途。这么做，甚至都不是在扩展软件的用途，而是增添与目的无关的功能。

同样重要的是要思考用户的需求。用户之所以要使用软件，总是有自己的需求。理想情况下，软件的用途应当相当接近用户的需求。举个例子，假设用户要的是计算自己的报税数据，那么他所需要软件的用途就是帮助用户计算个人的报税数据。

软件的用途如果不符合用户的需求，就很可能让用户的生活更复杂。如果用户希望的是阅读邮件，但是程序的主要目的是向用户展示广告，这两者就不协调。

想要看到用户勃然大怒吗？只需要刻意阻碍他们达到目的即可。在用户要做正事的时候，当面弹出几个窗口；或者新增一堆功能，让他无法分辨；或者显示一大堆他完全不认识的图标。让用户勃然大怒方法多多，都无非是干扰用户的需求，背离软件自身的基本用途。

有时候，营销人员或者经理会给软件设定一些目标，但是这些目标其实并不符合程序的基本用途，比如“要好玩一些”、“设计要更有冲击感”、“要受新媒体欢迎”，“要使用最新的技术”等等。这些人可能是公司里的重要人物，但他们不是那些决定程序应当做什么的人。身为软件设计师或是技术经理，你的职责是保证软件的基本用途，防止它偏离正轨，这个责任其他人谁也担不起。有时候，你可能需要为此据理力争，但从长期来看，这肯定是值得的。

而且，这不是要你为此去承担营销失利的责任。已经有许多产品因为执着于单一用途而获得了巨大的成功。肥皂的用途是清洁，食盐的用途是调味，灯泡的用途是照明，许多公司已经依靠这些产品生存了几十年。有效的营销与复杂的产品没有必然联系，要做好营销，你需要懂得营销的知识和技巧，但是这个领域与软件设计完全不同。

这是千真万确的，没有必要玩太多花样，做太复杂，尝试用单个软件瞬间完成 500 个任务。最受用户喜欢的软件是专注而简洁的，并且始终执着于基本用途。

8.2 糟糕的技术

出现复杂性的另一个常见原因就是，系统里选择了错误的技术，尤其是最终发现并不能很好适应未来需求的技术。但是既然无法预测未来，现在就决定要选择什么技术并不简单。好在，开始使用之前，你可以通过三个因素来判断技术是否“糟糕”：生存潜力、互通性、对品质的重视。

8.2.1 生存潜力

某种技术的生存潜力，就是它持续获得维护的可能性。如果某种类库或依赖项已经过时，没有人维护，你却死守着它们，就有麻烦在等着你了。

要了解某款软件的生存潜力，可以查阅其最近的发布记录。开发者是否频繁地发布真正解决了客户问题的新版本？另外，开发者对 bug 报告的响应有多积极？他们是否有活跃的邮件列表或支持团队？是否有足够的人在线谈论这项技术？如果某项技术现在有足够的活力，你可以相当肯定它不会很快灭亡。

另外还得看看，是否只有一家供应商在推进这项技术，以及它是否被不同开发人员开发的各种程序所接受和使用。如果只有一家供应商来推动和改造，那么万一供应商甩手不管或者决定停止维护，你就麻烦了。

接受广泛程度

这个说法听起来的意思似乎是，应当从符合需求的技术中选择接受最广泛的。从某种程度上说这是真的，被接受的技术一般都是有相当大的生存潜力。不过，你还是必须辨别，到底是经过考验才被大家接受，还是仅仅是因为某种垄断而被接受。

在写作本书时，C是经过验证的广泛接受的语言。在各种公司，有各种人，为各种不同的目的使用C语言。关于C语言有许多国际标准，每项标准又有众多的实现，包含许多广泛使用的编译器。

不过，有些技术之所以被广泛接受，是因为用户别无选择[1]。假设X公司设计了自己的编程语言，然后又设计了一种广泛接受的设备，这种设备只接受这种语言编写的程序。这就是上面提到的“单一供应商”情况，语言看起来被广泛接受，但生存潜力其实很小，除非它能被整个软件行业广泛接受。

8.2.2 互通性

所谓互通性，指的是如果需要，从一种技术切换到另一种技术有多难。要了解技术的互通性，就得问问自己：“我们能不能以某种标准方式来交互，这样就更容易切换到遵循同样标准的其他系统？”

举例来说，有些国际标准规定了程序应当如何与数据库系统交互。有些数据库对这种标准的支持很好，如果你选择这些支持良好的数据库，将来程序只需要做很小的改动，就很容易切换到其他数据库。

不过，也有些数据库对标准的支持并不好。如果你希望切换到这种支持糟糕的数据库，就得重写自己的程序。所以，如果选择非标准系统，就会身陷罗网，难以切换。

8.2.3 对品质的重视

这是一种更主观的衡量，其思想是考察最近的发布中，产品是否更加完善了。如果你可以看到源代码，检查一下开发人员是否进行了重构，清理了代码。产品是更容易使用了，还是更难用了？维护它的人真的在乎产品的质量吗？最近是否多次出现似乎由糟糕的编程所引发的严重安全问题？

注1：开发人员可能对自己所用的技术有独特的热情。为了避免触犯这类用户，我们还是不提具体的技术为佳。

8.2.4 其他原因

在选择技术时，还有其他因素需要考虑，主要是它是否简洁，是否符合软件的基本用途。在衡量完所有实际因素之后，也可以考虑个人喜好。有些人更看重某种编程语言在某些方面的优势。有时候，这也是选择技术的有效理由：如果两种技术在其他方面都差不多，就会选择更喜欢的那种。毕竟，最终真正使用它的人是你，你的意见很重要！上面这些指引只负责帮你排除掉糟糕的选择，剩下的则取决于你自己的研究、需求、计划。

8.3 复杂性及错误的解决方案

通常，如果某件事情变得非常复杂，也就意味绝不是表面的复杂那么简单，而是设计出了问题。

假如汽车的轮子是方形的，它当然开不快。这时候，改造发动机并不能解决问题，你需要重新设计汽车，换上圆形的轮子。

一旦程序里出现了“无法解决的复杂性”，就说明设计中有些深层次的基本错误。如果问题在这个层面上无法解决，应当回过头去看看产生问题的真正原因是什么。

其实，程序员经常这么做。你可能会说：这代码太垃圾了，要添加新功能真够麻烦的。那么，深层次的基本问题就是代码太混乱。所以，你应该整理这些代码，让它们变简洁，你就会发现增加新功能也会变简单。

真正要解决的问题是什么？

如果有人问你：“怎么让小马飞上月球？”你要反问的就是：“要解决的究竟是什么问题？”你可能发现，这个人真正需要的其实是收集一些灰色的岩石。为什么要去月球，还必须弄一匹小马，只有他自己知道。很多人真的会弄混淆这类问题，所以应当问问他们，究竟要解决

什么问题，然后简单的办法就会自己冒出来。在上面的例子里，一旦彻底弄清楚了问题，解决方案就很简单直接：去室外找些灰色的岩石就够了，根本用不着小马。

所以，如果事情变复杂，不妨回过头去看看真正要解决的是什么问题。你可以退上一大步，也可以质疑任何问题。有可能为了得到 4，你想到的是 2+2，但没想到 1+3 也可以，或者干脆不要加法，直接取得 4 也行。真正的问题是"我如何得到 4 这个数字"，任何可行的方案都是可以接受的。所以你真正要做的就是，找出自己所处的环境中最好的办法。

要做到这一点，不应当依靠猜测，而必须亲眼去看要解决的问题。确认你真正理解了问题的方方面面，找到最简单的解决办法。不要问"用现有代码怎么解决这个问题"，或者"Anne 教授在程序里是怎么解决这个问题的"，而是问问你自己："通常情况下，在最完美的方案里，这类问题要如何解决？"这样，你大概就可以看出代码应该如何重写了。然后，就可以着手解决问题。

再然后，问题就不复存在了。

8.4 复杂问题

有时候你受命去解决一些本身就非常复杂的问题，比如编写拼写检查或国际象棋的程序。问题复杂，解法不一定会复杂；相反，在处理此类问题时，你必须更努力地追求代码的简洁。

如果你在解决复杂问题时遇到了麻烦，那么用简单易懂的文字把它写在纸上，或者画出来。有些最优秀的程序设计就是在纸上完成的，真的。把它输入到计算机里只是次要的细节。

> **大多数麻烦的设计问题，都可以用在纸上画图或写出来的办法找到答案。**

8.5 应对复杂性

身为程序员，你必然要面对复杂性。其他程序员会写复杂的程序，你必须去修正。硬件设计师和语言设计师会让你的生活更麻烦。

如果系统中某个部分太过复杂，有个好办法来解决：把它分解成几个独立的小部分，逐步重新设计。每次修改都应该足够小，这样可以放心动手，不会让事情更复杂。不过这个过程中最大的危险是，新做的修改有可能会引入更多的复杂性。许多重设计或重写的工作之所以最终失败，就是因为它们引入了更多的复杂性，结果最后和原有系统同样复杂。

每个步骤都应该足够小，比如给某个变量取个更好的名字，或是给难看懂的代码增加一些注释。更常见的做法是在每个步骤中都把一个复杂的部分拆分成若干个简单的部分。

如果所有的代码都包含在一个巨大的文件里，改进的第一步就是把某个部分保存到单独的文件里。之后改进这个小部分的设计，然后把另一个部分保存到新的文件，再改进这个部分的设计。如此重复下去，最终得到的就是可靠的、可理解的、可维护的系统。

如果系统非常复杂，这么做的工作量可能相当大，所以必须有耐心。你必须首先做好设计，改进之后的系统要比现在的简单——即便只是简单一点点。然后，朝着这个目标一步步地前进。得到了这个简单的系统之后，就设计下一个更简单的系统，再朝那个目标前进。不必设计“完美”的系统，因为它不存在。你只需要持续不懈地追求比现有系统更好的系统，最终就会得到相当容易管理的简洁系统。

不过，还有一点也很重要，你不能专门花很长的时间来重新设计，停止开发新功能。变化定律告诉我们，程序所处的环境是持续变化的，所以程序的功能也必须去适应这些变化。如果在相当长的时间里，你都不能从用户角度出发来调适和改进，就可能失去自己的用户，把项目弄死。

好在平衡开发新功能和应对复杂性这两项任务的方法有很多。最好的一个办法就是，重新设计时只考虑让新功能更容易实现，然后实现这个功能。这样，你就可以在重新设计和开发新功能之间定期切换。它同样有助于保证新设计能够适应需求，因为设计时会考虑到实际的应用。系统的复杂性也会逐渐下降，而且你一直都跟得上用户的需求。你甚至可以这样来处理 bug：如果发现修改设计之后，某些 bug 更容易修复，那么先重新设计代码再修复 bug。

为新增功能重新设计

有个叫Bugzilla的项目，它的所有数据都存储在数据库中。Bugzilla只支持用某种特定的数据库来存储，这里我们叫它OldDB。有些新客户要求使用别的数据库来存储数据，我们叫它NewDB。客户的理由很充分：相比OldDB，他们更熟悉NewDB，而且他们公司已经在用NewDB了。但是，原有的客户希望继续使用OldDB。

所以，Bugzilla必须支持多种数据库。这需要大量修改代码，因为Bugzilla没有统一存取数据库的接口。相反，代码里散落着很多自定义的数据库处理指令，它们只适用于OldDB，而无法对接NewDB。一个解决办法是在代码中增加许多`if`语句，在每个访问数据库的地方区分NewDB和OldDB特殊处理。这样代码库的复杂性几乎会加倍，而Bugzilla的团队只有少数兼职程序员，要是系统的复杂性加倍，他们就维护不了了。

于是，Bugzilla团队决定采取另一个办法，重新设计系统，让它能容易支持多种数据库。这样的改动是个大工程。以下概要描述了他们的步骤。

(1) Bugzilla中对所有数据库系统已经有一套操作标准，但没有广泛使用。所以检查整个系统，每次修改一个文件，只要可能，就把它切换为标准操作。
(2) 对没有标准版本的数据库操作，写一个函数，它会返回适用当前所用数据库的正确操作。为每个非标准操作创建一个函数，把非标准操作替换为函数调用。重复这个过程，直到去掉所有的非标准函数。

(3) 项目中有许多代码服务于专属于OldDB的功能。停止使用这些OldDB专有的功能，切换到可适用于所有数据库的标准功能。每次只修改一个功能，如果需要，还可以分更细的步骤。
(4) 重新设计Bugziila的安装程序，让它可以设定任何数据库系统，而不只是OldDB。首先要重新设计安装程序，把它变简单，然后整理代码，使其同时支持OldDB和NewDB。

以上任何一步本身都是一个项目。它们被拆分为小的步骤，所以每一步的工作都得到了良好的设计。而且，系统做了任何改动之后都会进行测试，确保对OldDB的功能没有变化。

这样最后得到了一个完美的系统吗？没有，只是得到了比之前更好的系统——不但支持NewDB，而且更容易维护。最终Bugzilla能够支持4种不同的数据库，这全都是因为做了上面的工作后，它更容易支持新的数据库了。[2]

8.5.1 把某个部分变简单

上面说的都很不错，很有道理，但到底该做什么才能把某个部分变简单？嗯，这就要用到现有的关于软件设计的知识了。学习设计模式和设计方法来处理遗留代码，学习软件工程师普遍使用的工具，都可以帮上大忙，尤其有用的是掌握多门编程语言，熟悉多种不同的类库，因为每一种语言和类库都会用自己的方式来思考问题，即便你并不使用这些语言和类库，这种思维方式也可以应用到你的具体环境中。

掌握了这些知识，在面对复杂性问题时你就会有更多的选择。软件设计的法则可以帮助你选择不错的选项，在这之后，你的判断力和经验

注2：在过去许多年间，因为各种原因，Bugzilla 按照这个套路重新设计了很多次。如果你想知道已经完成的重要工作，可以查阅这里已经划掉的项：https://bugzilla.mozilla.org/showdependencytree.cgi?id=278579&hide_resolved=0。如果你想知道关于数据库的改动如何完成的更详细情况，请查阅这里已经划掉的项：https://bugzilla.mozilla.org/showdependencytree.cgi?id=98304&hide_resolved=0。如果你熟悉数据库系统，阅读各项的标题就大概知道整个项目是怎么完成的了。

可以决定针对当前的具体问题要做什么。不要仅仅因为权威的肯定就机械地生搬硬套某个工具，我们的选择永远是，在当前的环境下，当前的代码中，做合适的事情。

虽然有时会有这样的情况：你看到某一段代码，却不知道有什么工具可以化简它。或者你可能刚刚开始学编程，还没有时间马上研究这些信息。如果是这样，你应该看着这个复杂问题问自己：要怎么做，才可以让事情处理或是理解起来更容易？这就是在每个化解复杂性的问题背后的关键点。应对之道在于找到一种让代码更简单的可行办法。软件设计的工具和技巧只能锦上添花，帮助你找到更好的答案。

8.5.2 不可解决的复杂性

简化系统时，你可能会发现某些复杂性是无可避免的，可能所使用的硬件就是很复杂的。如果遇到这类不可解决的复杂性，你要做的就是屏蔽这种复杂性。在程序外面妥善包装上一层，让其他程序员更容易使用和理解。

8.6 推倒重来

面对非常复杂的系统，有些设计师会选择推倒重来。不过，把系统推到重来，几乎就是设计师承认失败，等于说："我们设计不出可维护的系统，所以只能重来。"

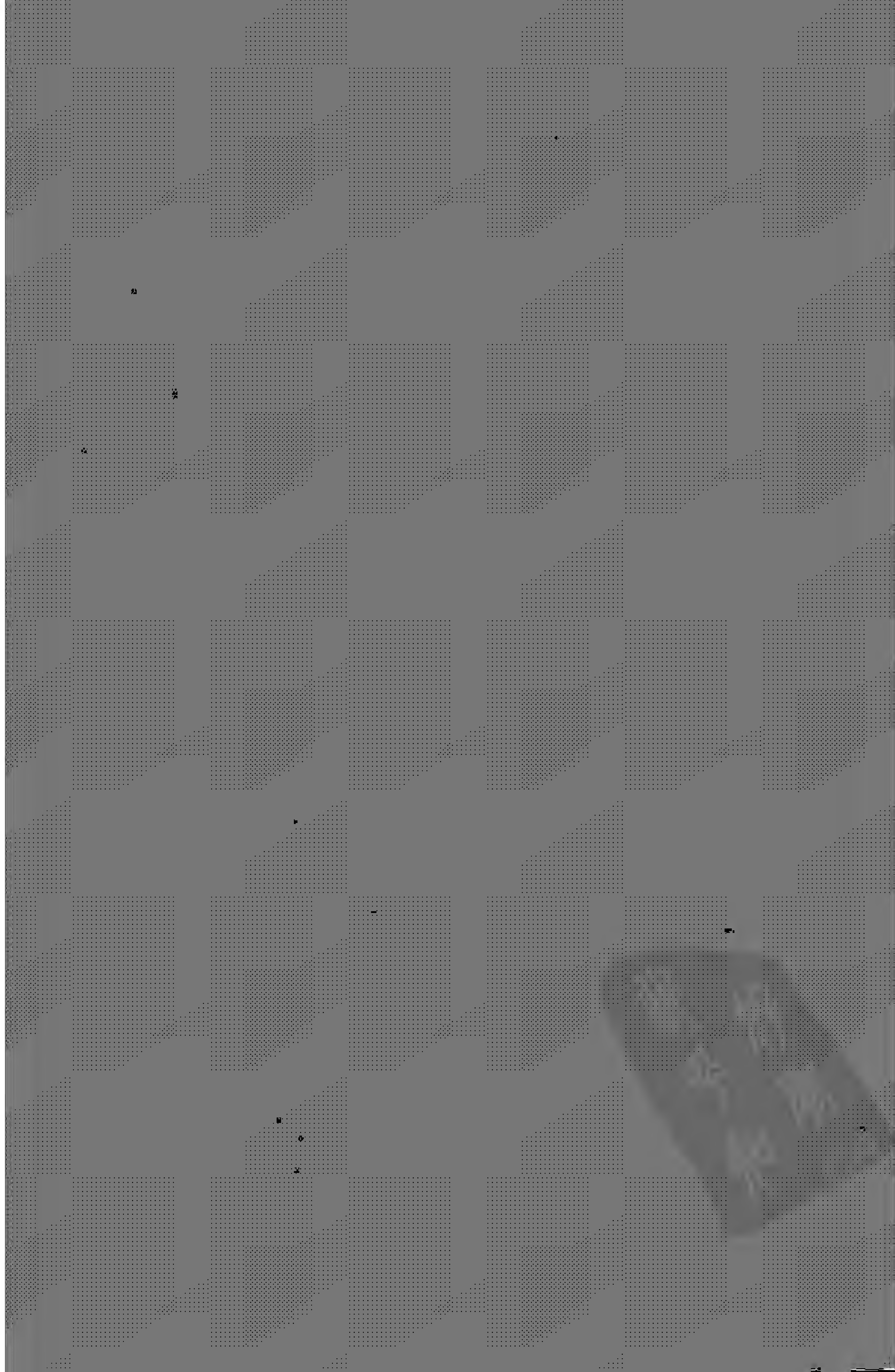
有人认为所有的系统最终都要重写，但事实不是这样的。系统在设计完成之后永远不会被抛弃是有可能做到的。软件设计师所说的"某天，因为某个原因，我们不得不抛弃所有代码"，就好像建筑师说的"这座摩天大楼应该在某天以某种方式倒塌"。如果摩天大楼设计有问题，后期也没有很好地维护，那么它必然会倒塌。可要是从一开始就建得很好，而且维护得很好，为什么会倒塌呢？

既然能建造出坚固稳定的摩天大楼，也就能构建出可维护的软件系统。

上面说了这么多，不过，还是有些时候重写是可以接受的。但是，这种情况非常罕见。如果下面的条件全都满足，你才应该重写。

(1) 你已经完成了准确评估，证明重写整个系统会比重新设计现有系统更有效率。只有猜测是不够的，你需要真正去做一些重新设计现有系统的试验，然后对比结果。已有的复杂系统可能很难应付，某些部分可能很难处理，但是为了知道修复它需要多少时间，你必须动手做一些尝试。
(2) 你有足够的时间用来开发新的系统。
(3) 你要比原有系统的设计师更高明，或者，如果原有系统是你设计的，但现在你的设计能力已经大大提升了。
(4) 你完全打算好了通过一系列简单的步骤设计整个新系统，在每一步都有用户提供反馈。
(5) 你有足够的资源，可兼顾维护原有系统和重新设计系统。绝对不要为了让程序员重写新系统而停止对原有系统的维护。系统只要在使用，都离不开维护。请记住，你自己的精力也是一种资源，必须慎重分配——如果两线作战，你每天有足够的时间分配给原有系统和新系统吗？

如果上面的条件都满足，那么推倒重来是可以接受的。否则，应该做的事情不是推倒重来，而是降低现有系统的复杂性，也就是通过一系列简单步骤来改进系统的设计。



第9章

测试



我们无法保证程序将来一直可以正常使用，只能保证目前它可以正常运行。甚至这次运行是正常的，而下次就可能出现问题。可能周围的环境发生了变化，所以程序无法运行；也可能你换了一台机器，所以无法运行。

不过，希望还是有的——我们并不会备受程序功能不确定的无穷煎熬，测试法则（Law of Testing）告诉我们：

你对软件行为的了解程度，等于你真正测试它的程度。

软件最后一次测试的时间距离现在越近，它可以继续正常运行的可能性就越大。软件在越多的环境里测试过，你就越确定它可以运行在这些环境中。上面所说的测试的“程度”，包括软件有多少个方面你曾经测试过，上一次测试是多久以前，在多少不同的环境下进行过测试。总的来说，你可以这么理解：

除非亲自测试过，否则你不知道软件是否能正常运行。

仅仅“能正常运行”，其实是非常模糊的，所谓的“能正常运行”是指什么呢？你在测试时真正知道的是，软件的行为是否符合预期，所以，你必须清楚地知道预期的行为。这个要求看来有点傻，而且是理所当然的，但是测试的关键就在于此。针对每项测试，你必须问一个非常具体的问题，得到一个非常具体的答案。问题可能是：“程序启动之后，用户会首先看到这个按钮，那么程序第一次运行的时候，用户如果按下这个按钮，会发生什么？”然后你会得到具体的答案，比如：“程序会弹出一个窗口，显示‘Hello, World’。”

这样，就有了一个问题，而且知道了答案。如果还有其他的答案，那么软件就是“不能正常运行”的。

有些行为非常难测试，你只能问：“如果用户这么做，程序会崩溃吗？”期望回答是“不会”。但是对设计良好的软件，在大多数情况下，测试之后你会得到具体得多的信息。



你还必须保证测试是准确的。如果测试显示程序的行为完全符合预期，事实却并非如此，或者测试显示程序崩溃了，其实它跑得好好的，那么这些测试都是不准确的。

最后，你必须观察测试的结果，确保它们是有效的。如果测试失败了，必须有办法让你知道这个情况，以及失败的具体原因。

测试可能因为太简单而被忽视。我们写完代码然后保存就完了，却忘记了看它是否能正常运行。可是，无论程序员多么聪明，也无论有多少理论数据来证明你的代码是正确的，在没有切实测试过之前，你都不知道它能不能正常工作。

只要你修改了软件的某个部分，这部分是否能正常工作就成了未知数，就必须重新测试。而且，这部分很可能关联到其他许多方面，所以你不知道关联的部分是否会受影响。如果是大刀阔斧地改，你可能就需要重新测试整个程序。

显然，你不会希望每次进行了一点小修改之后，还要手工测试整个程序。所以如今的程序员在落实测试法则时，会编写许多自动化测试来测试所写的每一段代码。这样做的好处在于，有任何改动都只需要重新运行这些测试即可，程序会测试系统中的方方面面，确保修改之后所有部分仍然保持正常。

网上有许多资料讲解如何编写自动化测试以及测试的一般原理，也出版过很多相关书籍。测试领域的资料丰富详尽，值得学习。测试法则只负责解释为什么需要测试，什么时候需要测试，以及测试真正能提供什么信息。

附录A

软件设计的规则



本附录总结了书中讨论的所有可行的规则。

(1) 软件的目的是帮助他人。

(2) 软件设计的方程式是：

$$D = \frac{V_n + V_f}{E_i + E_m}$$

其中：

D 表示变化的合意程度（可行性）；

V_n 表示当前价值；

V_f 表示未来价值；

E_i 表示开发成本；

E_m 表示维护成本。

这是软件设计的主要法则。随时间的推移，这个方程式简化为：

$$D = \frac{V_f}{E_m}$$

也就是说，相比降低实现成本，降低维护成本更重要。

(3) 变化定律：程序存在的时间越久，它的某个部分需要变化的可能性越大。

(4) 缺陷定律：在程序中新增缺陷的可能性与代码修改量成正比。

(5) 简洁定律：软件任何一部分的维护难度，反比于该部分的简洁程度。

(6) 测试定律：你对软件行为的了解程度，等于你真正测试它的程度。

就是这么多。本书中讨论了很多的事实和想法，但这 6 条是软件设计的法则。请注意，其中最重要的是要牢记软件的目的、软件设计方程式的简化形式以及简洁定律。

如果你希望把最重要的事实综合成软件设计时要记得的两句话，就是：

- 相比降低开发成本，降低维护成本更加重要。
- 维护成本正比于系统的复杂程度。

有了这两条，以及对软件目的的了解，再加上你知道整个系统的复杂性源自各部分的复杂性，你就有相当的把握去重新认识软件设计这整门科学。